普通高等教育“十三五”规划教材

Python 程序设计基础

钟雪灵　李　立　主编

高平安　李梅生　唐名华　副主编

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

本书介绍 Python 程序设计的基础知识，集教材、习题、上机实验于一体，内容涵盖全国计算机等级考试二级 Python 语言程序设计的考试大纲，包括 11 章教学内容和 9 个实验。在教学内容中，第 1～6 章介绍 Python 程序设计的基础知识，第 7～9 章分别介绍三个用于数据分析的第三方程序包（NumPy、Pandas、Matplotlib）及其应用，第 10～11 章主要围绕 Tushare 财经数据库介绍运用 Python 进行数据分析的几个综合案例。9 个实验与教学内容结合紧密，包括验证性、设计性和综合性实验，方便实验教学的组织与开展。

本书可作为高等院校计算机专业和非计算机专业零编程基础学生的教材，也可作为全国计算机等级考试的教材，还可以作为相关人员的自学参考书。

图书在版编目（CIP）数据

Python 程序设计基础/钟雪灵，李立主编．—北京：电子工业出版社，2019.12
ISBN 978-7-121-37595-8

I. ①P… II. ①钟… ②李… III. ①软件工具－程序设计－高等学校－教材 IV. ①TP311.561

中国版本图书馆 CIP 数据核字（2019）第 219792 号

责任编辑：谭海平
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×1092　1/16　　印张：18.75　字数：480 千字
版　　次：2019 年 12 月第 1 版
印　　次：2019 年 12 月第 1 次印刷
定　　价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）88254552，tan02@phei.com.cn。

前　言

数字时代已经来临。移动互联网、云计算、大数据、人工智能等先进信息技术层出不穷，不断渗透至社会的各个领域，产生了许多新的应用场景，深刻地改变着人们的社交方式、生活方式和工作方式。一些智者认为构建在现实世界基础之上的数字化虚拟世界已逐渐形成。如今，人们不仅生活在有形的现实世界，而且生活在现实与虚拟相互交织、相互融合的数字时代。

在数字时代，具备一定的计算思维，能够理解计算机的工作原理，能够自然地实现人机交互，已成为年轻一代大学生的基本素养。显然，缺乏这些素养的人将难以理解区块链、大数据、人工智能、物联网等不断涌现的先进技术，更无从谈起它们的应用和创新。因此，令人担忧的是，缺乏这些素养的人将游离于数字时代之外，无法理解时代的变化，更无法适应这种变化，最终将面临被时代淘汰的危险。为此，数字时代的高等教育必须加强大学生信息技术能力的培养，新一代大学生无论学习什么专业、未来从事何种职业，都必须掌握一些信息技术，熟练使用一些工具。掌握一门计算机语言是新一代大学生开启信息技术学习大门的钥匙，而掌握数据分析是开启大门后最有价值的探究方向之一。数据分析是指通过信息技术方法和手段从数据中发现知识、预测未来，它是数字时代一门通用的基础技术，应用范围广，作用力强。创业者通过数据分析可以优化产品，营销人员通过数据分析可以改进营销策略，产品经理通过数据分析可以洞察用户真实需求，金融从业者通过数据分析可以规避投资风险……因此，大学生学好一门计算机语言，继而结合自身专业学习程序化的数据分析方法，是提高信息技术能力最直接、最有效的一条学习路径。

近年来，Python 已迅速成为数据分析和机器学习领域最主流的计算机语言，有着 Excel、SPSS、R 等工具无可比拟的多种优势。Python 是一种跨平台、免费、开源、面向对象的解释型高级程序设计语言，遵循简单、明确、优雅的理念，具有易写、易读、易维护、拓展性强等特点。Python 提供了丰富的标准程序包和第三方程序包，如功能强大的数据分析包 Pandas、NumPy、Matplotlib 等，在程序中导入包后就可使用该包提供的所有功能。

广东金融学院是一所具有“央行基因”的财经类院校，为提高学生的信息技术能力，多年前就已开设“Python 程序设计基础”课程，编制了丰富的教学讲义和实验素材。为进一步践行我们的人才培养理念，我们组织经验丰富的教师团队针对财经类专业编写了本书，力图为学生进一步学习数据分析打下扎实的语言基础。本书包括 11 章教学内容和 9 个实验。在教学内容中，第 1～6 章介绍了 Python 程序设计的基础知识；第 7～9 章分别介绍三个用于数据分析的第三方程序包（NumPy、Pandas、Matplotlib）及其应用；第 10～11 章主要围绕 Tushare 财经数据库介绍运用 Python 进行数据分析的几个综合案例。9 个实验与教学内容结合紧密，包括验证性、设计性和综合性等类型实验，方便实验教学的组织与开展。此外，本书内容涵盖了全国计算机等级考试二级 Python 语言程序设计的考试大纲，可作为二级考试的教材。

本书由钟雪灵、李立、高平安、李梅生和唐名华五位老师共同编写。在教材编写过程中，潘

章明、侯昉和陈灵三位老师提供了无私的帮助和支持。在此一并向为本书出版付出辛勤劳动的朋友们表示衷心的感谢！

最后，若读者对本书有任何意见或建议，请发送电子邮件至 tzhongxl@gduf.edu.cn，以便在图书再版时完善。对此我们表示由衷的感谢！

钟雪灵

2019 年 10 月 1 日于广州

目　录

第 1 章　初识 Python

Python 是一门跨平台、免费、开源、面向对象的解释型高级程序设计语言。它简单易学、用途广泛，是目前最受欢迎的高级程序设计语言之一。本章介绍的内容如下：Python 语言的发展历史和特点；Python 安装包的下载与安装；使用 pip 工具管理 Python 软件包；使用 Anaconda 软件管理 Python 软件包；IPython 开发工具和 Spyder 集成开发环境。

1.1　Python 语言概述

Python 语言是由荷兰人 Guido van Rossum 于 1989 年设计并实现的高级程序设计语言。自 1991 年公开发布 Python 0.9.0 以来，分别于 1994 年、2000 年和 2008 年相继发布了 Python 1.0、Python 2.0 和 Python 3.0。相对于此前的版本，Python 3.0 的改变较大，Python 3.0 不完全兼容之前的版本，在 Python 2.0 环境下正常运行的程序未必能在 Python 3.0 环境下运行。Python 3.x 是目前得到广泛应用和强大技术支持的版本系列，Python 3.7 是当前最新的 Python 3 版本。

自第一版公开发行以来，Python 开发者和用户社区都在不断壮大，Python 已成为非常流行的热门程序开发语言。Python 在 2010 年是所有编程语言中市场份额增速最快的语言，并因此获得 Tiobe 2010 年度编程语言大奖。

Python 是一门跨平台、开源、免费的解释型计算机程序设计语言。Python 支持命令式编程和部分函数式编程，也完全支持面向对象程序设计，并拥有大量的功能扩展库。它可以把多种不同语言编写的程序融合到一起实现无缝拼接，从而更好地发挥不同语言和工具的优势，更好地满足不同应用领域的需求。

Python 已被广泛应用于诸如数据分析、组件集成、网络服务、图像处理、数值计算和科学计算等众多领域。目前业内几乎所有大中型互联网企业都在使用 Python，如谷歌、百度、腾讯、汽车之家、美团等。互联网公司广泛使用 Python 程序实现的应用主要包括自动化运维、自动化测试、大数据分析、爬虫及 Web 应用等。

1.2　Python 的开发优势

Python 是目前最流行且发展最迅速的计算机语言之一，它的简洁性使得软件的代码得到大幅缩减，开发任务得到显著简化。高效的高级数据结构、优美的语法和动态类型，以及语言的解释型特性，使得 Python 成为快速应用开发的理想语言。Python 语言有许多重要特性，而且有些特性富有显著创造性。

（1）简单易学

相对于其他语言，Python 语言摈弃了复杂结构、简化了语法、减少了关键字，使得 Python 程序结构更简单、语法更清晰，可阅读性更强，也使得初学者更容易掌握 Python 编程方法。

（2）免费开源

开源（即开放源代码）意味着任何人都可以得到软件的源代码，而且可以修改并替换源代码。

Python 是遵守 FLOSS（Free/Libre and Open Source Software，自由/开源软件）规范的语言，用户可以免费获取 Python 的源代码进行研究，甚至可对它进行二次开发。正因为 Python 是开源软件，所以不断有许多优秀程序员加入 Python 的开发行列，将自己的源代码添加到 Python 中，从而使 Python 的功能日臻丰富和完善。

（3）跨平台性和可移植性

由于开源本质，无须任何修改，Python 程序就可在很多平台上运行。这些平台包括 Windows、Mac OS、iOS 及各种 Linux/Unix 系统。例如，在 Windows 系统上编写的 Python 程序，可以很方便的移植到 iOS 或 Android 系统上运行。

（4）解释型语言

计算机只能理解二进制形式的机器语言，而不能直接理解 Python 这类高级语言。因此，编译型语言（如 C 语言）的源程序需要编译成机器语言才能在计算机中执行。但 Python 语言是解释型语言，其源程序不需要编译成二进制代码，而只需要 Python 解释器把源程序转换成字节码，然后再把字节码翻译成计算机使用的机器语言就可以运行。这就使得使用 Python 更加简单，也使得 Python 程序更加易于移植。

（5）面向对象

Python 既支持面向过程的编程，又支持面向对象的编程。“面向过程”的语言程序是由过程或可重用代码的函数构建的。“面向对象”的语言程序由数据和功能组合而成的对象构建而成。

（6）可扩展性

由于 Python 可以调用 C/C++代码，所以若需要一段关键代码运行得更快或希望某些算法不公开，那么该部分程序可以采用 C 或 C++语言编写，再嵌入 Python 程序。

（7）丰富的库

Python 具有庞大的标准库。这些库可以帮助处理包括正则表达式、文档生成、单元测试、线程、数据库、网页浏览器、CGI、FTP、电子邮件、XML、XML-RPC、HTML、WAV 文件、密码系统、GUI（图形用户界面）及其他与系统有关的操作。除标准库以外，Python 还拥有许多辅助工具库，如 wxPython、Twisted 和 Python 图像库等。

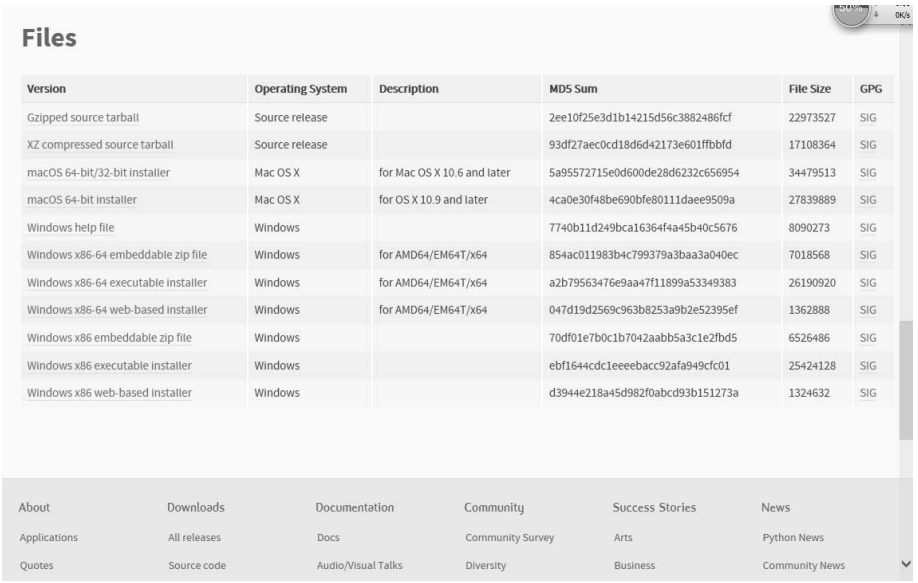
1.3 安装 Python

Python 是跨平台的，可在多个平台上运行。我们可以在 Python 的官网下载 Python 安装程序，目前的最新版为 Python 3.7.3，下载链接为 https://www.python.org/downloads/release/python-373/，网站上提供了多种类型的安装包，如图 1.1 所示。下面以 Windows 系统为例介绍 Python 的安装。

鼠标双击下载的 Python 安装包，进入 Python 安装向导，出现如图 1.2 所示的 Python 安装方式选择界面。

在如图 1.2 所示的界面中单击 Install Now 链接开始执行安装操作，屏幕上会出现 Python 安装进度界面。Python 安装完成后，系统会显示如图 1.3 所示的 Python 安装成功界面。

在如图 1.3 所示的界面中单击 Close 按钮结束安装过程。



Files

Version	Operating System	Description	MD5 Sum	File Size	GPG
Gzipped source tarball	Source release		2ee10f25e3d1b14215d56c3882486fcf	22973527	SIG
XZ compressed source tarball	Source release		93df27aec0cd18d6d42173e601ffbbfd	17108364	SIG
macOS 64-bit/32-bit installer	Mac OS X	for Mac OS X 10.6 and later	5a95572715e0d600de28d6232c656954	34479513	SIG
macOS 64-bit installer	Mac OS X	for OS X 10.9 and later	4ca0e30f48be690bfe80111daee9509a	27839889	SIG
Windows help file	Windows		7740b11d249bca16364f4a45b40c5676	8090273	SIG
Windows x86-64 embeddable zip file	Windows	for AMD64/EM64T/x64	854ac011983b4c799379a3baa3a040ec	7018568	SIG
Windows x86-64 executable installer	Windows	for AMD64/EM64T/x64	a2b79563476e9aa47f11899a53349383	26190920	SIG
Windows x86-64 web-based installer	Windows	for AMD64/EM64T/x64	047d19d2569c963b8253a9b2e52395ef	1362888	SIG
Windows x86 embeddable zip file	Windows		70df01e7b0c1b7042aabb5a3c1e2fbd5	6526486	SIG
Windows x86 executable installer	Windows		ebf1644cdc1eeeebacc92afa949cfc01	25424128	SIG
Windows x86 web-based installer	Windows		d3944e218a45d982f0abcd93b151273a	1324632	SIG

About　Downloads　Documentation　Community　Success Stories　News
Applications　All releases　Docs　Community Survey　Arts　Python News
Quotes　Source code　Audio/Visual Talks　Diversity　Business　Community News

图 1.1　Python 安装程序下载页面

图 1.2　Python 安装方式选择界面

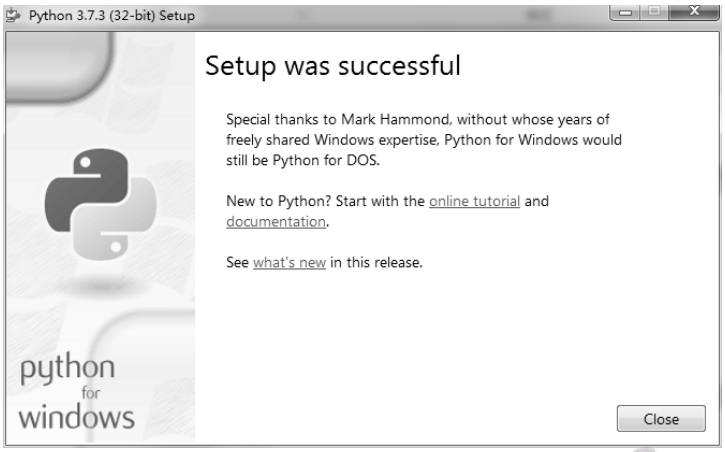


图 1.3　Python 安装成功界面

1.4 IDLE 开发环境

1.4.1 Python IDLE 集成开发环境

Python IDLE（Integrated Development and Learning Environment）是一个集成开发环境。它提供 Python 程序代码的编辑、解释、调试、运行等诸多功能的一体化服务。

启动 Python 集成开发环境 IDLE 的操作步骤如下：选择 Windows 的“开始”菜单→Python 3.7 程序组→IDLE (Python 3.7 32-bit)菜单项，打开如图 1.4 所示的 Python 集成开发环境窗口。该窗口也是 Python 的命令（Shell）窗口，用户可以在窗口中编写并运行 Python 程序。

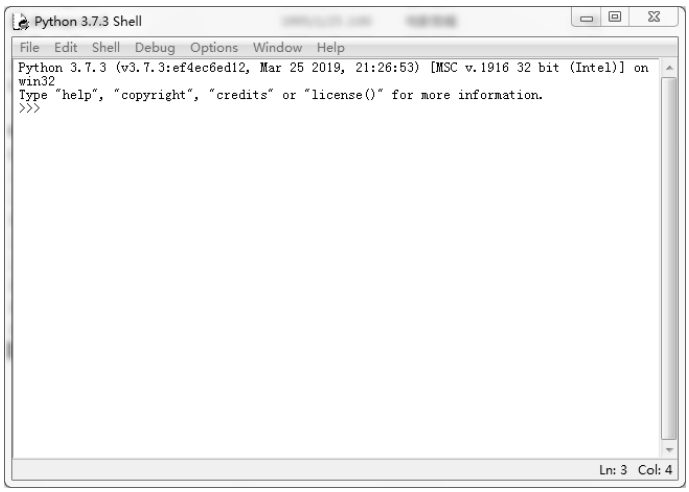


图 1.4 Python 集成开发环境窗口

1.4.2 Python 程序编辑器

IDLE 自带程序编辑器窗口，可以在程序编辑器中对 Python 源程序进行编辑，实现源程序的输入、修改等操作。下面介绍 Python 程序编辑器窗口的相关操作。

1．新建程序文件

在如图 1.4 所示的 Python 集成开发环境窗口中选择 File 菜单→New File 命令或按组合键 Ctrl+N，打开一个如图 1.5 所示的 Python 程序编辑器窗口，窗口的标题为程序文件名，初始文件名 untitled 表示该文件是还未命名保存的 Python 程序。

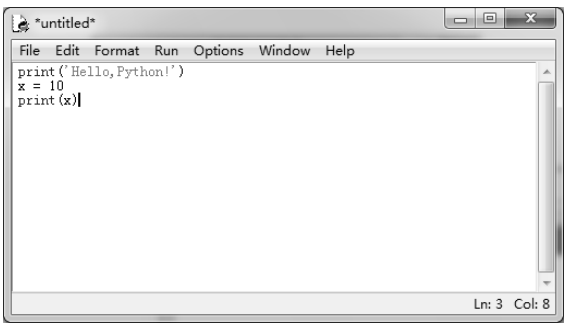


图 1.5 Python 程序编辑器窗口

2．保存程序文件

在如图 1.5 所示的 Python 程序编辑器窗口中输入程序代码，选择 File 菜单→Save File 命令或按组合键 Ctrl+S，将程序代码保存为 Python 源文件。如果程序代码是第一次保存，那么系统会弹出保存文件的对话框，在对话框中输入要保存的文件名（文件的扩展名为“.py”），然后单击对话框中的“保存”按钮完成保存操作。

3．打开程序文件

在图 1.4 所示的 Python 集成开发环境窗口中或图 1.5 所示的 Python 程序编辑器窗口中选择 File 菜单→Open File 命令，或按组合键 Ctrl+O，就会弹出打开文件对话框。用户在对话框中选择需要打开的.py 文件，然后单击对话框中的“打开”按钮，就会在 Python 程序编辑器窗口中打开程序文件。

也可右键单击.py 源程序的文件名，在弹出的快捷菜单中选择 Edit with IDLE 命令，将该文件在 Python 程序编辑器窗口中打开。

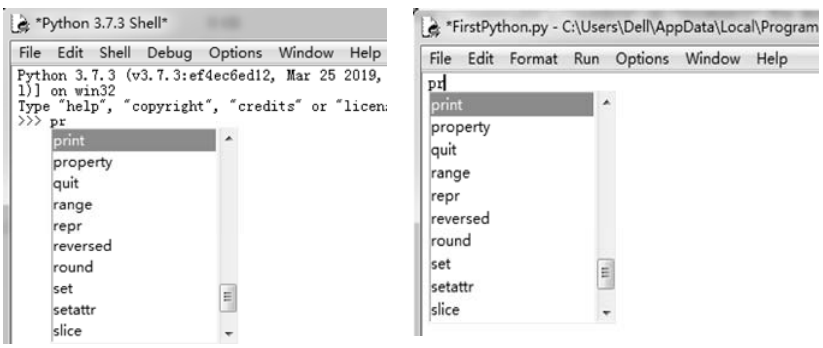
4．语法高亮

Python IDLE 支持语法高亮，编辑器能够以彩色标识程序代码中的 Python 关键字、用户定义的函数名和字符串等内容以便提示程序员。例如，在 Python IDLE 中注释显示为红色，函数显示为紫色，字符串显示为绿色。

5．自动完成

自动完成是指输入命令单词的开头部分后，IDLE 会根据 Python 语法或上下文自动完成单词后面部分的填写。执行自动完成的操作步骤如下：在程序编辑器中选择 Edit 菜单→Expand Word 命令，或按组合键 Alt+/。自动完成将查找编辑器内已经写过的代码来自动补全。例如，如果前面的语句中已经输入 print，那么后续语句中可在输入“pr”后按组合键 Alt+/，完成 print 的自动输入。按组合键 Alt+p 可调出上一条历史命令，按组合键 Alt+n 可调出下一条历史命令。

也可以首先只输入 Python 保留字或函数名的开头部分，然后在 IDLE 或程序编辑器中按 Tab 键，打开如图 1.6 所示的提示框，在提示框列表中选择 print，完成 print 的自动输入。



(a) IDLE 命令窗口　　(b) Python 程序编辑器窗口

图 1.6　Python 的自动完成

1.4.3　Python 程序书写规范

Python 程序的解释系统要求程序的书写格式符合约定的规范。Python 源程序的书写规范主要体现在语句的格式、语句块与缩进、注释等方面。

（1）语句

程序语句是程序要求计算机完成某个具体运算的指令。原则上要求 Python 程序的每一行只写一条语句。如果非要在一行内写多条语句，那么必须用分号“;”隔开不同的语句。如果一条语句过长，那么可以使用字符“\”将其分割为多行。

【例】多条语句写在同一行：

```
a = 10; b = 20; c = 30
```

上面一行程序代码包含了“a = 10”、“b = 20”和“c = 30”三条不同的语句，所以必须使用分号将它们隔开。

【例】一条语句占多行：

```
content = '在 Python 源程序中如果一行内写多条语句，\
语句间用分号（;）隔开。\
如果一条语句过长，可以使用反斜杠符号实现多行书写。'
```

由于这个赋值语句比较长，因此将它分割成 3 行，前面两行的末尾都使用了分行字符“\”来表示语句被分割成多行。

（2）语句块与缩进

语句块是指多个语句构成的复合语句，它是程序中完成相对复杂功能且逻辑独立的语句组合体。Python 程序中的语句块必须使用缩进方式表示，这种缩进要体现不同语句块之间的层级关系。

【例】Python 的语句块与缩进：

```
a = 20
b = 10
if  a > b:    # 语法要求，此处需有冒号
    c = a     # 下面三条语句都要缩进，且缩进要一致
    a = b
    b = c
print(a, b)
```

上面的程序使用了 if 语句判断 a 与 b 的大小。如果 a 的值大于 b 的值就交换 a 和 b 的值，实现交换操作的三条语句（分别是“c = a”、“a = b”和“b = c”）构成一个语句块，因此这个语句块以缩进的形式体现其与 if 语句的从属关系。

关于代码的缩进，需要注意以下几点：

- 语句块相对上级语句应缩进。
- 不要混用空格和 Tab 键实现缩进。
- 缩进是分级的，同一级别的缩进必须对齐，否则会导致语法错误。

由于语句块中可以嵌套语句块，因此程序中可以存在多级缩进，并通过每行代码前面的空格数来识别缩进级别。不同的空格数表示不同的级别，同一级别语句的缩进必须相同。例如，在上面的代码中，“c = a”、“a = b”和“b = c”三条语句的级别相同，因此缩进深度一致。标准 Python 风格中每个缩进级别是 4 个空格（在 IDLE 中按 Tab 键会自动将 1 个制表符转为 4 个空格）。

（3）注释

注释用于描述程序或语句的功能，其目的是增强程序的可读性。Python 允许在源程序中添加注释，在执行程序时会忽略程序中的所有注释。Python 的注释分为单行注释和多行注释两种。单行注释从字符“#”开始，到该行结束为止。多行注释使用 3 个引号（单引号或双引号）作为注释的开始和结束标记。

【例】Python 的注释：

```
# -*- coding: utf-8 -*-
"""
Created on Mon May 13 15:09:15 2019

@author:
"""
print('Hello,Python!')      # 输出显示文字 Hello,Python!
```

上面的程序中，第一行使用的单行注释独占一行，第二行以 3 个英文双引号（"）作为多行注释的开始，第六行的 3 个双引号表示这个多行注释的结束，最后一行在 print 语句后面使用“#”给出了一个单行注释。

1.4.4　Python 程序的运行

1．Python 程序的运行原理

用 Python 语言编写的程序被称为 Python 源程序（也称源代码）。计算机运行 Python 源程序时需要使用 Python 解释器对源程序进行解释。从计算机的角度来看，Python 程序的运行过程包含两个步骤：首先由解释器将源代码解释为字节码（也称中间码），然后由虚拟机执行字节码。Python 程序运行的基本流程如图 1.7 所示。

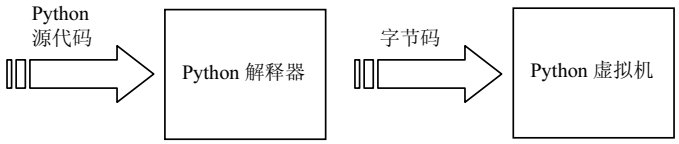


图 1.7　Python 程序运行的基本流程

Python 源程序（保存后的程序文件的扩展名为“.py”）经过 Python 解释器翻译成字节码，然后由 Python 虚拟机（Python Virtual Machine，PVM）逐条将字节码翻译成机器指令执行。

2．Python 程序的运行方式

运行 Python 程序的方式有交互方式和文件方式两种。交互方式也称命令（Shell）方式，是指 Python 解释器即时响应并执行用户输入的程序代码。文件方式（即编程方式）是指用户将 Python 源代码以程序文件方式保存，然后由 Python 解释器批量执行文件中的程序代码。交互方式适合运行或调试少量代码，如果需要实现相对复杂的功能，那么就应以文件方式运行 Python 程序。

（1）运行 Python 程序的交互方式

在如图 1.8 所示的 IDLE 命令提示符“>>>”后面输入单行 Python 代码，输入后按 Enter 键就会直接交互执行。例如，在提示符“>>>”后面输入代码“print('Hello,Python!')”后按 Enter 键，Python 解释器就会立即执行输入的代码，并将执行结果“Hello,Python!”显示在下一行。

在 Python 交互方式中，每行代码均以 Enter 键结束。若有输出，则显示该代码的运算结果。图 1.8 所示代码中的 print()是一个用于输出数据的函数，后面会详细介绍 print()函数。

如果在命令提示符“>>>”后面输入 exit()或 quit()函数，那么计算机执行命令后就会退出 Python 运行环境。

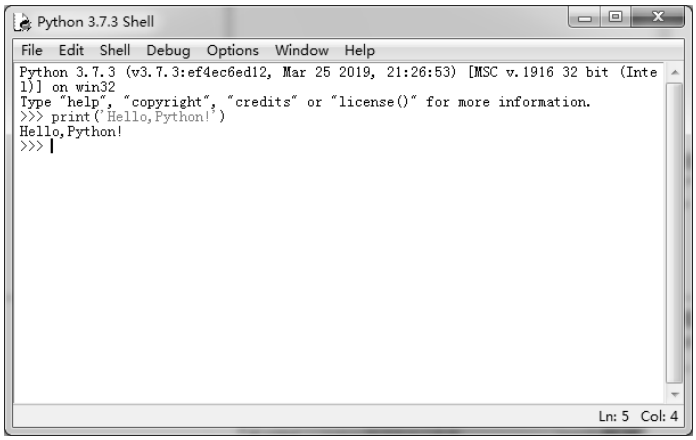


图 1.8　运行 Python 程序的交互方式

（2）运行 Python 程序的文件方式

打开如图 1.5 所示的程序编辑器窗口，输入程序代码。编辑完程序代码后，选择菜单栏中的 File 菜单→Save 命令保存代码，例如将其存储为 FirstPython.py。然后，选择菜单栏中的 Run 菜单→Run Module 命令（或按组合键 F5），执行当前编辑器中的程序代码，程序代码的运行结果将在 IDLE 中显示（见图 1.9）。如果运行代码时出错，那么会以红色文字显示错误信息以提示用户。

用户可以使用其他编辑器（如记事本）编辑 Python 程序的代码，但必须以“.py”为扩展名保存程序文件。在菜单栏中选择 File 菜单→Open 命令可以打开已有的程序文件，然后编辑或运行该文件中的代码。

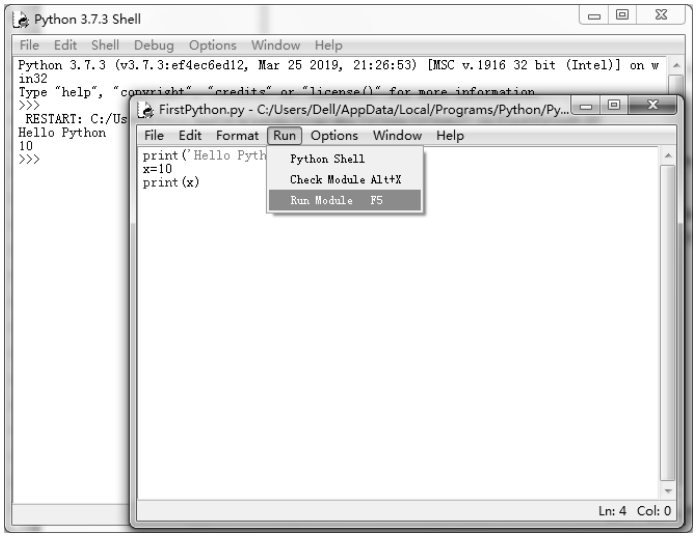


图 1.9　运行 Python 程序的文件方式

1.5　Anaconda 开发环境

1.5.1　Python 软件包管理工具

Python 的强大之处体现在它的应用领域十分广泛。Python 得到广泛应用的关键原因是它拥

有数量庞大且功能完善的标准库和第三方库。为各个领域开发的软件包存放在网络软件仓库中，用户需要时通过网络下载合适的软件包，安装到计算机中即可使用，极大地提高了软件开发者的编程效率。

PyPI（Python Package Index）是 Python 官方的第三方库，官网链接为 https://pypi.org/，所有人都可以下载或上传 Python 软件包到 PyPI 库。Python 用来下载、安装和管理公共资源库 PyPI 软件包的管理程序主要有 easy_install 和 pip。easy_install 程序提供了自动搜索指定软件包合适版本、安装软件包及配置各种文件的功能；pip 是 easy_install 的改进版，它进一步简化了 Python 安装第三方库的操作。下面主要介绍 pip 的安装和使用。

1．pip 工具的下载与安装

安装 Python 时一般都自带了 pip 工具。如果已经安装了 Python，那么在 Windows 操作系统中进入命令提示符窗口，在命令行中输入“pip --version”命令即可判断本机是否已安装 pip；若已经安装 pip，则会显示它的版本。注意命令中 version 的前面是两个“-”字符。

如果本机没有安装 pip 工具，那么可以按照下面的步骤进行安装：

步骤 1：在官网 https://pypi.org/project/pip/上下载 pip 安装包。

步骤 2：解压 pip 安装包，解压后的文件夹中应含有一个 setup.py 文件。

步骤 3：在 Windows 的命令提示符窗口中进入存放 setup.py 的文件夹，然后在命令提示符后输入“python setup.py install”安装 pip。

2．pip 工具的使用

pip 提供了对 Python 软件包的查找、下载、安装和卸载功能。

（1）查看已安装的 Python 软件包

在命令提示符窗口中输入“pip list”命令可以查看 Python 已安装了哪些软件包，结果如图 1.10 所示。

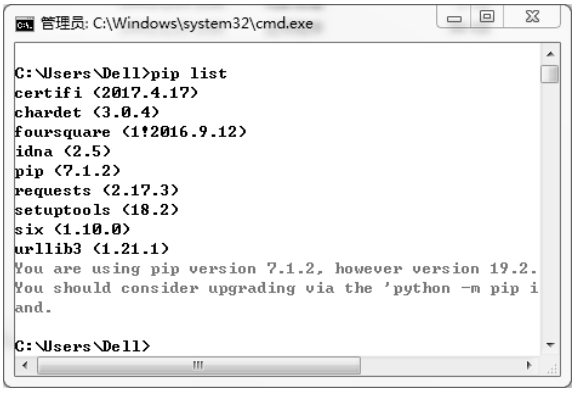


图 1.10　查看已安装的 Python 软件包

（2）查看可升级的 Python 软件包

输入“pip list --outdated”命令可以检查已安装的 Python 软件包中哪些包是可升级的（其中 outdated 前面有两个“-”字符）。

（3）安装 Python 软件包

输入“pip install 软件包名”命令可以安装指定的 Python 软件包，pip 会自动在网络上搜索并安装适合本机 Python 版本的软件包。例如：

```
c:\> pip  install   numpy        # 安装 NumPy 科学计算库
c:\> pip  install   pillow       # 安装 pillow 图片处理库
```

（4）更新 Python 软件包

输入“pip install --upgrade 软件包名”命令可以更新已安装的 Python 软件包（upgrade 前面有两个“-”字符）。

（5）卸载 Python 软件包

输入“pip uninstall 软件包名”命令可以卸载已安装的 Python 软件包。

（6）查看软件包的详细信息

输入“pip show 软件包名”命令可以查看已安装的 Python 软件包详细信息。例如：

```
c:\>pip  show  numpy   # 查看已安装的 NumPy 库的相关信息
```

1.5.2 Anaconda 软件包

Python 自带的集成开发环境（IDLE）可以编辑和运行 Python 程序，但其功能相对简单。虽然通过 pip 可以安装、管理和维护 Python 软件包，但由于 Python 的标准库和第三方库数量庞大，管理和维护这些软件包是一件比较麻烦的事情。Anaconda 是一个易于实现 Python 软件包和虚拟环境管理与维护的 Python 开发平台。目前的 Anaconda 包含了 300 多种最受欢迎的 Python 软件包。下面介绍 Anaconda 程序的下载与安装。

可以在 Anaconda 的官网 https://www.anaconda.com/distribution/下载安装程序。官网提供了适合 Windows、Mac OS 和 Linux 等多种操作系统的安装包，以及适合不同 Python 版本的安装包。下面介绍在 Windows 操作系统环境下安装 Anaconda 的基本过程。

双击下载的 Anaconda 安装程序启动安装向导，在如图 1.11 所示的 Anaconda 安装欢迎界面单击 Next 按钮，进入 Anaconda 权限许可界面，如图 1.12 所示。

在图 1.12 所示的界面单击 I Agree 按钮，进入如图 1.13 所示的 Anaconda 安装类型选择界面，此时可以选择 Anaconda 软件仅限本用户使用（Just Me 选项），也可以选择所有用户都能使用（All Users 选项）。选择安装类型后，单击界面上的 Next 按钮进入 Anaconda 安装位置设定界面，如图 1.14 所示。

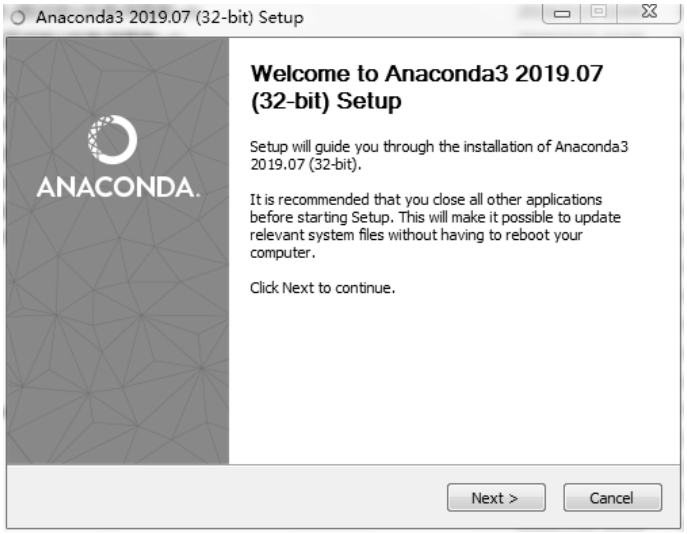


图 1.11 Anaconda 安装欢迎界面

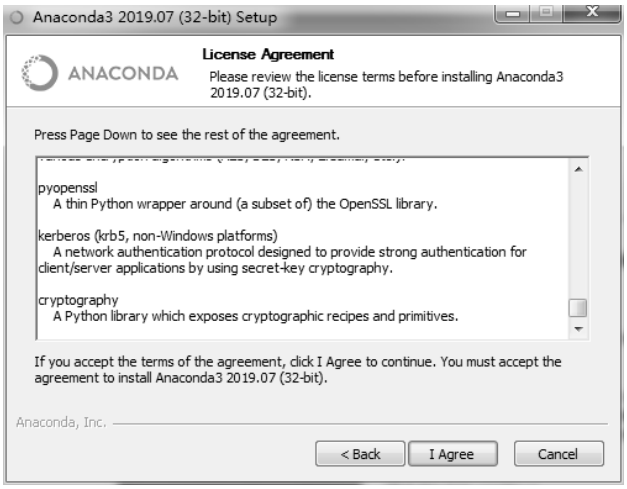


图 1.12　Anaconda 权限许可界面

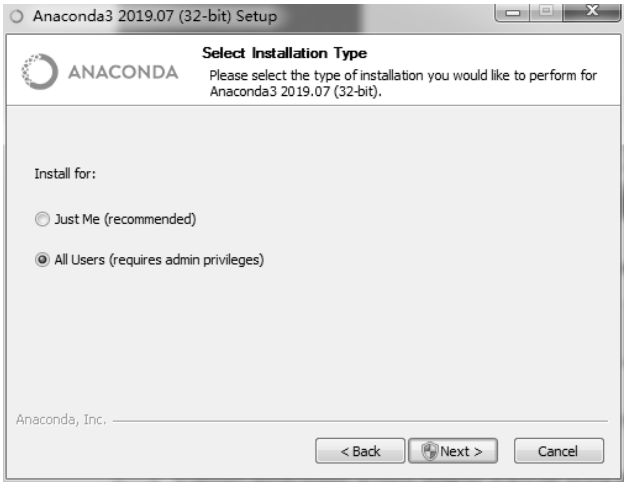


图 1.13　Anaconda 安装类型选择界面

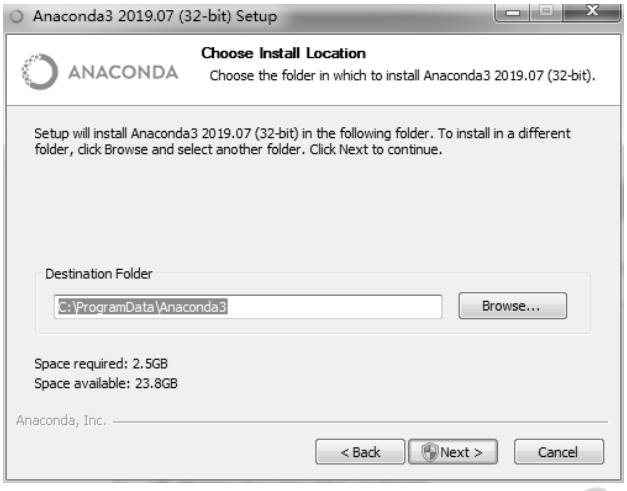


图 1.14　Anaconda 安装位置设定界面

Anaconda 默认的安装位置是 C:\ProgramData\Anaconda3，用户可根据自己的需要修改安装位置。选择好位置（文件夹）后，单击 Next 按钮开始安装 Anaconda，按照安装向导提示一直单击各界面上的 Next 按钮，直到出现如图 1.15 所示的 Anaconda 安装结束界面，此时单击界面上的 Finish 按钮结束安装过程。

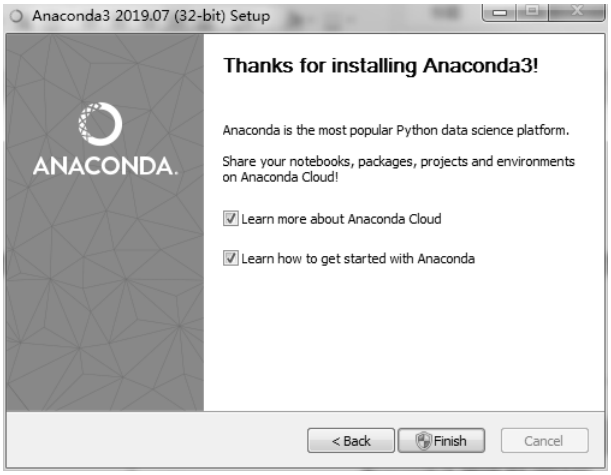


图 1.15　Anaconda 安装结束界面

Anaconda 平台中包含了常用的 Anaconda Prompt 窗口、IPython 开发工具、Spyder 集成开发环境和 Jupyter Notebook 开发环境等。下面简单介绍 Anaconda Prompt、Ipython 和 Spyder 的使用。

1.5.3　Anaconda Prompt 窗口

Anaconda Prompt 是 Anaconda 的命令提示符窗口，它类似于 Windows 系统的命令提示符窗口，在此窗口中可以输入各种 Anaconda 的配置命令，以完成类似 Python 软件包管理和虚拟环境管理的操作。

1．Anaconda Prompt 窗口的启动

启动 Anaconda Prompt 窗口的步骤如下：选择 Windows 的“开始”菜单→Anaconda3 程序组→Anaconda Prompt 命令，打开如图 1.16 所示的 Anaconda Prompt 窗口。

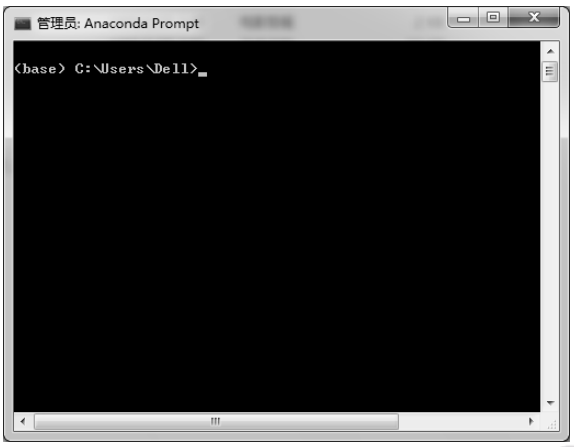


图 1.16　Anaconda Prompt 窗口

2．Anaconda 软件包管理

在 Anaconda Prompt 窗口中可方便地使用 conda 工具对 Python 软件包进行安装、更新和删除等操作。conda 管理软件包的常用命令格式如表 1.1 所示。

表 1.1　conda 管理软件包的常用命令

命　令	描　述
conda list	显示当前环境下的所有 Python 包
conda list -n envname packagename	显示 envname 环境中名为 packagename 的包
conda install packagename	在当前环境安装名为 packagename 的包
conda install -n envname packagename	在 envname 环境中安装名为 packagename 的包
conda search packagename	搜索名为 packagename 的包
conda update packagename	更新当前环境中名为 packagename 的包
conda update -n envname packagename	更新 envname 环境中名为 packagename 的包
conda remove packagename	删除当前环境中名为 packagename 的包
conda remove -n envname packagename	删除 envname 环境中名为 packagename 的包

例如，在 Anaconda Prompt 窗口输入“conda list”命令可以查看当前环境下已安装的 Python 包，系统显示结果如图 1.17 所示。

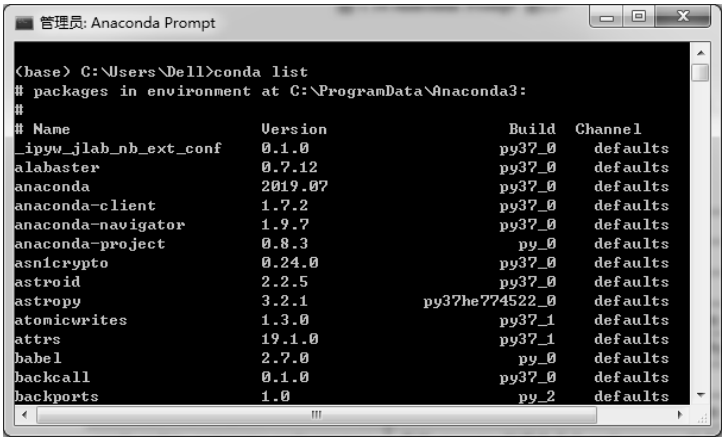


图 1.17　查看已安装的 Python 包

3．Anaconda 环境管理

用户可以在 Anaconda 中建立多个虚拟环境，每个虚拟环境具有独立的 Python 运行环境。因此利用虚拟环境可以隔离不同项目所需的不同版本的软件包，从而避免不同版本之间的冲突。例如，由于 Python 3 不兼容 Python 2，因此可以分别建立 Python 2 和 Python 3 两个虚拟环境，以运行不同版本的 Python 程序。

在 Anaconda Prompt 窗口中可以使用 conda 工具实现虚拟环境的管理，包括查看虚拟环境、创建虚拟环境、删除虚拟环境和切换虚拟环境等操作。

（1）创建虚拟环境

创建新的虚拟环境的命令格式如下：

```
conda create -n env_name python=version package_names
```

-n：参数，指定后续的虚拟环境名，也可用--name 代替。

env_name：虚拟环境名。

python=version：为虚拟环境指定 Python 版本。

package_names：可选项，在创建新的虚拟环境时要安装的软件包。

【例】创建一个 Python 3.7 虚拟环境，其虚拟环境命名为 python37：

```
conda create --name python37 python=3.7
```

（2）查看虚拟环境

查看 Python 的虚拟环境的命令格式如下：

```
conda env list
```

在 Anaconda Prompt 窗口输入命令“conda env list”查看 Python 的虚拟环境的结果如图 1.18 所示，其中的 base 是安装 Anaconda 时默认安装的 Python 环境，符号“*”指当前使用的 Python 环境。

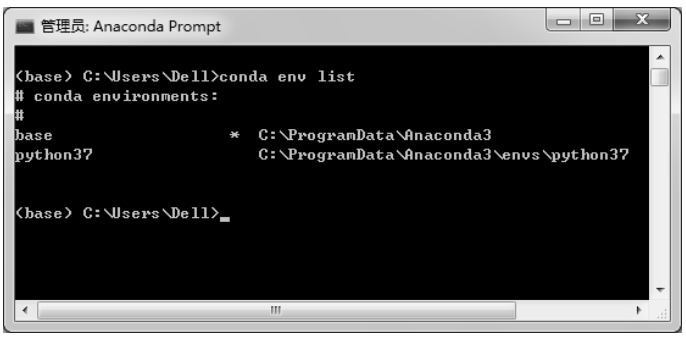


图 1.18 查看 Python 虚拟环境

（3）切换虚拟环境

改变当前运行的虚拟环境的命令格式为

```
conda activate env_name
```

env_name：虚拟环境名。

【例】从当前环境“base”切换到刚创建的“python37”环境：

```
conda activate python37
```

执行这个命令会即刻切换到虚拟环境 python37，此时在 Anaconda Prompt 窗口的命令行前面会出现 python37 而非 base 提示符。

（4）退出虚拟环境

退出当前虚拟环境时并不需要指定虚拟环境名，其命令格式如下：

```
conda deactivate
```

（5）删除虚拟环境

删除指定虚拟环境的命令格式如下：

```
conda remove -n env_name --all
```

或

```
conda env remove -n env_name
```

env_name：虚拟环境名。

1.5.4 IPython 开发工具

IPython 是一个增强的交互式 Python shell，已成为现代科学计算中最重要的 Python 工具。它支持变量自动补全和自动缩进，集成了 Python 调试器、%run 机制、创建多个环境及调用系统 shell

等诸多实用功能，是进行科学计算和交互可视化的一个最佳 Python 平台。

1．IPython 交互式命令窗口的启动

在 IPython 中，可以用交互模式运行所有输入的 Python 源码。IPython 交互式命令窗口的启动有下面几种方法。

方法 1：启动 Anaconda Prompt 窗口，在窗口中输入“ipython”命令，进入 IPython 交互式命令窗口，如图 1.19 所示。

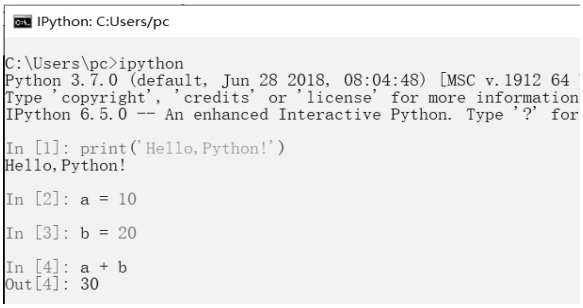


图 1.19　IPython 交互式命令窗口

方法 2：选择 Windows 的“开始”菜单→Anaconda3 程序组→Spyder 命令，打开如图 1.20 所示的 Spyder 集成开发环境窗口。Spyder 集成开发环境集成了 IPython 开发工具，在窗口的“控制台窗格区”中提供了 IPython 交互命令窗格，用户可以在此窗格中编辑并执行 Python 程序。关于 Spyder 集成开发环境的使用将在下一节介绍。

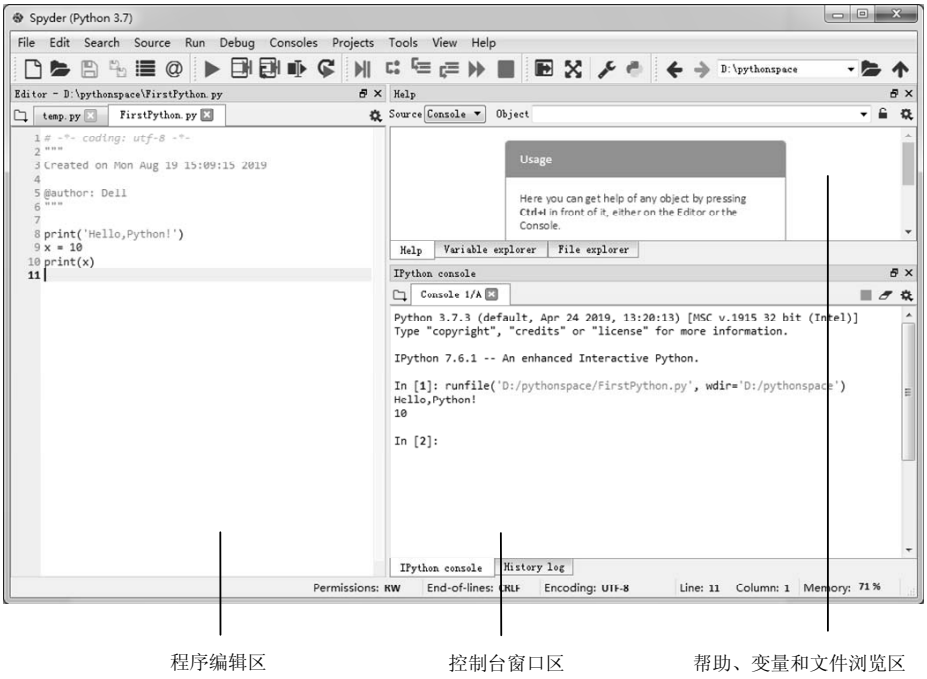


图 1.20　Spyder 集成开发环境

Python IDLE 以“>>>”作为输入提示符；IPython 窗口采用“In[序号]:”提示符表示程序代码的输入位置，使用“Out[序号]:”提示符表示执行结果的显示输出。

要特别说明的是，本书后面的程序主要以 IPython 作为调试与执行工具。如果没有明确声明，那么后面的章节中会用“In:”和“Out:”分别描述 IPython 窗口中的源代码输入和结果输出，即省略了“[序号]”。IPython 的“In:”后可一次连续执行多条语句，而 IDLE 的“>>>”提示符后一次只能输入执行一条语句。

2．IPython 的简单使用

（1）Tab 键自动补全

如果在 IPython 命令窗口中输入代码时按下 Tab 键，那么系统会自动列出与当前输入字符串匹配的内容（对象或函数）。例如，如果用户在输入“pr”后按下 Tab 键，那么系统会自动列出以“pr”开头的对象或函数（见图 1.21），此时使用键盘上的方向键选择列表中的选项后便自动完成输入。

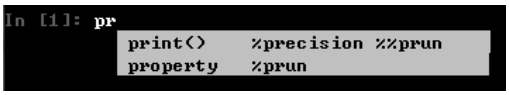


图 1.21　Tab 键自动补全

（2）使用“_”符号显示输出结果

在 IPython 中可以使用下画线（“_”）输出最近的显示结果。“_”与序号组合可以输出指定的显示结果，具体格式如表 1.2 所示。

表 1.2　“_”输出显示结果

格　式	描　述
_	显示最近的第一个输出结果
__	显示最近的第二个输出结果
_序号	显示指定序号的输出结果
_i 序号	查看指定序号的输入表达式

【例】使用“_”输出最近的显示结果：

```
In [1]: a = 20
In [2]: b = 10
In [3]: a + b
Out[3]: 30
In [4]: a - b
Out[4]: 10
In [5]: a * b
Out[5]: 200
In [6]: a / b
Out[6]: 2.0
In [7]: _                    # 显示最近的第一个输出结果
Out[7]: 2.0
In [8]: __                   # 显示最近的第二个输出结果
Out[8]: 2.0
In [9]: _5                   # 显示序号为 5 的输出结果
Out[9]: 200
In [10]: _i5                 # 显示序号为 5 的输入表达式
Out[10]: 'a * b'
```

需要注意的是，在“_i 序号”中“i”和序号之间不能有空格；“_序号”只能是 Out 中的序号，而“_i 序号”既可以是 Out 中的序号，又可以是 In 中的序号。

（3）历史记录功能

为了更好地利用以前的输入，IPython 会记录每次输入的命令。使用上下箭头键可调出历史命令。在 IPython 中使用 history（或 hist）命令可以查看历史输入内容，具体格式如表 1.3 所示。

表 1.3　history 命令格式

格　式	描　述
history	输出命令的历史记录
history -n 序号	输出指定序号的命令记录
history -g 字符串	搜索内容

【例】使用 history 命令查询历史输入：

```
In [1]: a = 20
In [2]: b = 10
In [3]: a + b
Out[3]: 30
In [4]: a - b
Out[4]: 10
In [5]: history                  # 显示所有历史输入
a = 20
b = 10
a + b
a - b
history
In [6]: history -n 2             # 查询序号为 2 的输入
   2: b = 10
```

（4）自省

在 IPython 中可以使用符号“?”查看系统已有的命令、变量或函数的信息，这种功能被称为自省。在命令、变量或函数的前面或后面输入“?”即可使用自省功能。

【例】使用“?”获取变量的信息：

```
In [1]: a = 10
In [2]: a?
Type:        int
String form: 10
Docstring:
int([x]) -> integer
int(x, base=10) -> integer
Convert a number or string to an integer, or return 0 if no arguments
```

在函数的前面或后面使用符号“??”可以查看函数的源代码。此外，使用如下命令也可显示变量 a 的相关信息，其中 dir()和 help()都是 Python 内建的帮助函数：

```
In [3]: dir(a)
In [4]: help(a)
```

（5）magic 命令

IPython 有一类被称为 magic 命令的魔术命令。IPython 将任何首字母为%的行视为 magic 命令。

magic 命令可以为常见的任务提供便利，也可以控制 IPython 系统的行为，从而为 IPython 增加许多系统功能。常用的 magic 命令如表 1.4 所示。

表 1.4 常用的 magic 命令

命 令	描 述
%quickref	显示 IPython 的快速参考
%magic	显示所有魔术命令的详细文档
%history	打开命令的输入（可选输出）历史
%reset	删除当前命名空间中的全部变量和名称
%run 源程序文件	执行 Python 源程序
%prun statement	通过 cProfile 执行 statement，并打印分析器的输出结果
%time statement	报告 statement 的执行时间
%timeit statement	多次执行 statement 以测试其平均执行时间，适合测试执行时间很短的代码
%who, %who_ls, %whos	显示命名空间中定义的变量

【例】magic 命令的使用。

（1）使用%history 查看历史输入：

```
In [1]: a = 20
In [2]: b = 10
In [3]: a + b
Out[3]: 30
In [4]: %history            # 查看历史输入
a = 20
b = 10
a + b
```

（2）使用%run 运行 Python 程序文件：

```
In [1]: %run  d:\pythonspace\FirstPython.py
Hello,Python!
10
```

（3）使用%timeit 测试运行时间：

```
In: %timeit  list(range(100))    # 测试 list(range(100))的运行时间
914 ns ± 6.09 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

1.5.5 Spyder 集成开发环境

Spyder 是一个用于科学计算的 Python 集成开发环境。它结合了集成开发工具的高级编辑、分析、调试功能，以及数据探索、交互式执行、深度检查和数据可视化功能，为用户带来了很大的便利。

1.5.4 节中介绍了启动 Spyder 的方法。一般情况下，Spyder 集成开发环境窗口主要包含程序编辑区、控制台窗格区、帮助、变量和文件浏览区等（见图 1.20）。Spyder 的程序编辑区提供以文件形式实现源程序的输入、修改的操作，控制台窗格区中的 IPython 交互命令窗格可以交互式执行 Python 源码，帮助区、变量显示区和文件浏览区提供查看帮助文档、显示内存变量和磁盘文件列表等内容的视图。在 Spyder 中既可以采用交互方式编程，又可以使用文件方式编程。

下面介绍在 Spyder 中实现文件方式编程的方法。

1．新建程序文件

启动 Spyder 集成开发环境窗口后，在程序编辑区默认打开的程序文件为 temp.py。如果要新建一个程序文件，那么有如下几种方法：

方法 1：选择菜单栏中的 File→New file 命令。

方法 2：单击工具栏中的“新建文件”按钮。

方法 3：按组合键 Ctrl+N。

此时将在程序编辑区创建一个临时命名为 untitled0.py 的程序文件，第一次新建的临时文件名 untitled 后面的数字为 0，以后依次为 1, 2, …。

2．打开程序文件

可以在 Spyder 窗口打开并编辑已保存的 Python 程序文件。打开程序文件的操作步骤如下。

步骤 1：选择菜单栏中的 File→Open 命令，或单击工具栏中的“打开文件”按钮，或按组合键 Ctrl+O 键，弹出“打开文件”对话框。

步骤 2：在对话框中选择需要打开的程序文件后，单击“打开”按钮。

此时在程序编辑区中显示该程序文件的源代码（见图 1.20）。Spyder 的程序编辑区可以同时打开多个程序文件。

也可以在 Windows 资源管理器中选择需要打开的程序文件，然后将文件拖到 Spyder 的程序编辑区中打开该文件。

3．编辑程序文件

在 Spyder 中新建程序文件，或打开已有的 Python 程序，程序文件的代码都将在程序编辑区显示。在程序编辑区可以按照 1.4.3 节介绍的程序书写规范对源程序进行编辑。

4．运行程序文件

要运行程序，可以选择菜单栏中的 Run→Run 命令，或单击工具栏中的“执行文件”按钮，或按快捷键 F5 键。程序执行结果将在控制台窗口区显示。若程序有错误，则控制台窗口区中会显示错误信息。

需要注意的是，临时命名的程序文件在运行前必须要先保存才能执行，运行已经命名的程序文件将会自动保存源程序。

1.6　本章小结

1. Python 是一种跨平台、简单易学、开源、免费、可移植、易扩展的解释型高级编程语言，广泛用于数据分析、组件集成、网络服务、图像处理、数值计算和科学计算等众多领域。
2. Python 安装包可在 Python 官网下载，网站上提供了多种类型的安装包。Python 自带了一个集成开发环境（IDLE）。在 IDLE 中可完成程序的编辑、调试和运行等操作。
3. Python 源代码以“.py”为扩展名的文件保存。程序运行时，Python 源程序首先由 Python 解释器翻译成字节码，然后由 Python 虚拟机逐条将字节码翻译成机器指令执行。
4. 运行 Python 程序的方式有两种，分别为交互方式和文件方式。交互方式是指 Python 解释器即时响应并执行用户输入的程序代码；文件方式是指首先将 Python 源代码以程序文件方式保存，然后由 Python 解释器批量执行文件中的代码。交互方式一般在运行或调试少量代码时使用，文件方式则是常用的编程方式。

5. Anaconda 是一个易于对 Python 软件包和虚拟环境进行管理的平台，其中包含了 300 多种最受欢迎的科学计算、数据处理和数据分析类的 Python 软件包。Anaconda 平台中还包含了常用的 Anaconda Prompt 窗口、IPython 开发工具和 Spyder 集成开发环境。
6. 可以使用 pip 工具或 conda 工具实现对公共资源库 PyPI 中相关资源包的下载、安装和管理操作。

习题

一、单项选择题

1. 下列应用领域，不适合使用 Python 开发的是（　　）。
   A．科学计算　　B．网络服务　　C．实时处理　　D．数据库编程
2. Python 语言属于（　　）。
   A．机器语言　　B．汇编语言　　C．高级语言　　D．以上都不是
3. 关于 Python 语言，下面说法正确的是（　　）。
   A．属于编译型语言　　B．属于解释型语言
   C．属于自然语言　　D．以上都不正确
4. Python 程序文件的扩展名是（　　）。
   A．.python　　B．.pn　　C．.py　　D．.pt
5. （　）不属于 Python 的特性。
   A．面向对象　　B．免费开源　　C．低级语言　　D．可扩展性
6. Python 内置的集成开发环境是（　　）。
   A．PyCharm　　B．Pydev　　C．IDLE　　D．pip
7. 下列哪一项未包含在 Anaconda 软件包中（　　）。
   A．PyCharm　　B．IPython　　C．Spyder　　D．Jupyter Notebook
8. 使用 conda 工具安装软件包的命令是（　　）。
   A．`conda install packagename`　　B．`conda remove packagename`
   C．`conda search packagename`　　D．`conda update packagename`
9. 在 IPython 开发工具中按下（　　）键可实现自动补全。
   A．Esc　　B．Tab　　C．F1　　D．Backspace
10. 在 IPython 开发工具中，使用（　　）命令运行 Python 程序文件。
    A．`%magic`　　B．`%history`　　C．`%run`　　D．`%time`

二、简答题

1. 简述 Python 的特性。
2. 程序的编译方式和解释方式有何区别？
3. Python 应用领域主要有哪些？
4. 简述 Python 程序的运行原理。
5. 运行 Python 程序的方式有哪些？
6. 简述使用 pip 工具和 conda 工具管理软件包的命令。
7. IPython 开发工具中使用“_”符号显示输出结果的格式有哪些？

第 2 章　Python 语言基础

本章介绍 Python 语言的基本语法、标识符的命名规则，重点介绍 Python 的数据类型、变量、运算符与表达式、输入与输出，以及常见的内置函数和模块等内容。

2.1　基本数据类型

数据是客观事物的符号表示，在计算机科学中是指所有能输入计算机并被计算机程序处理的符号的总称。为了满足实际应用的需要，Python 将数据划分成多种类型，包括数值、字符串、列表、元组、字典等。本节主要介绍数值型和字符串等 Python 的基本数据类型，其中数值型包含整型（int）、浮点型（float）、复数型（complex）和布尔型（bool）等。由于程序对不同类型数据的运算方法和存储表示存在明显差别，因此在程序设计过程中务必要为待处理的数据确定合适的数据类型和有效的处理方法。

2.1.1　数值型

整数类型简称为整型（int），它是 Python 中常用的一种数据类型。整数有正整数、零和负整数，其含义与数学中的含义相同。Python 用 int 数据类型表示整型。

Python 语言用浮点型（float）表示数学中的实数（也称浮点数），它是既有整数部分又有小数部分的数。浮点数用 float 表示。

复数是由实数部分和虚数部分构成的数。Python 将复数型（complex）数值表示为 x+yj，其中 x 是实数部分（实部），y 是虚数部分（虚部），j 为虚数单位。

```
In: x = 10          # 整型
In: y = 3.14        # 浮点型
In: z = 1+2j        # 复数型
In: type(x), type(y), type(z)    # 用 type()函数测试三个变量的数据类型
Out: (int, float, complex)       # 分别为整型、浮点型、复数型
```

2.1.2　逻辑型

布尔型（bool）也称逻辑型，是用来表示逻辑值的数据类型。逻辑值在程序中用于条件真假逻辑判断，它只有“真”和“假”两个取值，其中“真”用 True 表示，“假”用 False 表示。

Python 中的逻辑型也可视为整型的子类，True 和 False 对应的整型值分别为 1 和 0。因此，Python 的逻辑型数据支持普通整型的所有计算，如“False + 1”是合法的计算。

另外，Python 中的 0 或空集（空列表、空元组和空字典等）对应的逻辑值都是 False。

```
In: x = 10
In: x > 5
Out: True
In: y = False
In: type(y)
Out: bool     # y 为 bool 型
```

```
In: y + 1     # y 为 False，可视为 0
Out: 1
```

2.1.3 字符串

Python 采用字符串来表示和存储文本(如中英文字符)。字符串是最常用的数据类型，Python 中的字符串是用成对的单引号（'）、双引号（"）或三引号（'''或"""）界定的一串字符，其中单引号和双引号用于界定单行文本，三引号通常用于界定多行文本。

【例】下面都是内容为“Python 程序设计”的字符串。

```
In:  'Python 程序设计'       # 单引号作为定界符
Out: 'Python 程序设计'

In:  "Python 程序设计"       # 双引号作为定界符
Out: 'Python 程序设计'
```

1．转义字符

转义字符是一些有特殊含义且难以用一般方式表达的字符，如回车符、换行符等。为了表达这些特殊字符，引入了一些特殊的定义。需要在字符串中表达特殊字符时，Python 用反斜杠（\）开头，后面紧跟一个字符来表示某个特定的转义字符，如表 2.1 所示。

表 2.1 Python 的转义字符

转义字符	描　述
\（行尾）	续行符
\\	反斜杠符号
\'	单引号
\"	双引号
\a	响铃
\b	退格（Backspace）
\000	空
\n	换行
\v	纵向制表符
\t	横向制表符
\r	回车
\f	换页
\0yy	八进制数，yy 代表八进制数，例如\012 表示十进制数 10
\xyy	十六进制数，yy 代表十六进制数，例如\x1a 表示十进制数 26
\other	其他字符以普通格式输出

2．多行字符串

Python 可以使用三引号作为多行字符串的定界符。

【例】包含了 3 行的字符串。

```
In: '''Python
       程序
       设计'''
Out: 'Python\n 程序\n 设计'
```

也可在用单引号和双引号作为定界符的字符串中，在行尾用续行符（\）实现多行字符串。

【例】使用续行符（\）实现多行字符串。

```
In: 'Python\
     程序\
     设计'
Out: 'Python 程序设计'
```

需要注意的是，使用三引号实现多行字符串时得到的字符串中包含了换行符（\n），而使用续行符时在字符串中不会包含换行符（\n）。

3．两种格式化方法：%和 format

字符串格式化是将数据按照某种格式要求（如长度、类型等）转换为字符串。在 Python 中，字符串格式化有两种方式，一种是“%”格式，另一种是“format”格式。

```
In: x=3.457;  y=10
In:'x=%.2f  y=%d' % (x, y)            # %格式
Out: 'x=3.46  y=10'
In:'x={:.2f} y={}'.format(x,y)        # format 格式
Out: 'x=3.46 y=10'
```

（1）%格式

%格式出现在较早版本的 Python 中。使用%格式化字符串时，以一个字符串作为模板。模板是由普通字符串和包含%的格式符组成的，这些%格式符为真实值预留位置，并规定真实值应该呈现的格式。模板中%格式符的格式如下：

```
% [flag] [width] [.precision] type
```

flag：为预留值设置对齐方式，可省略，对齐方式如表 2.2 所示。

表 2.2　对齐方式

符　号	说　明
-	左对齐
+	右对齐并在正数前面显示加号（+）
0	右对齐并在显示的数字前面填充 0
空格	右对齐

width：指定字符串的输出宽度，可省略。

precision：指定数值型数据保留的小数位数，可省略。

type：指定的格式符，不可省略，具体的格式符如表 2.3 所示。

表 2.3　格式符含义

符　号	说　明
%c	格式化字符及其 ASCII 码
%s	字符串
%d	整数
%o	无符号八进制数
%x	无符号十六进制数（小写）
%X	无符号十六进制数（大写）
%f	小数，可指定小数点后的精度
%e	用科学记数法格式表示
%g	根据值的大小决定使用%f 或%e

【例】使用%格式化字符串。

```
In: '%5d'  %  2          # 默认右对齐
Out: '    2'             # 前面补空格，共占 5 位字符宽度
In: '%-5d'  %  2         # 左对齐
Out: '2    '             # 后面补空格
In: '%05d'  %  2         # 右对齐且前面补 0
Out: '00002'

In: '%-10d %d'  % (12, 34.56)    # 此行有两个数据，需要将两个数据放在()中
Out: '12        34'              # 第 0 个数据左对齐（10 位宽），第 1 个数据取整

In: '%f' % 3.14                  # 浮点数格式
Out: '3.140000'

In: '%6.2f' % 3.1415927          # 指定宽度为 6，小数部分 2 位
Out: '  3.14'
```

（2）format 格式

format 格式是 Python 2.6 以后各版本中提供的字符串格式化方法，它比%格式化的能力更强、更灵活，Python 官方也推荐使用 format 方式格式化字符串。

```
In: r = 4.5
In: 'r={} , area={:.2f}'.format(r, 3.14*r*r)
Out: 'r=4.5 , area=63.59'
```

format 的语法格式如下：

```
<格式字符串>.format(<参数>)
```

<格式字符串>：普通字符串和{替换格式符}组成的字符串。

<参数>：匹配替换的内容。

format 的作用是将信息进行格式化输出。格式字符串包含普通字符串和替换字段字符串，替换字段是用大括号{}界定的内容。输出时 format()函数的参数值会替换格式字符串中与之匹配的替换字段。参数与替换字段的匹配关系有下面几种情况。

- 序号匹配

format()函数中的参数序号默认从 0 开始。如果替换字段名以数字序号的形式给出，那么它将与 format()函数对应序号上的参数进行匹配。如果格式字符串的{}中没有指定序号，那么将按照格式字符串中{}出现的先后次序与实际参数进行匹配。

【例】通过序号匹配字符串。

```
# {}中指定了序号
In: '{0}和{1}都是流行的程序设计语言'.format('Python', 'Java')
Out: 'Python 和 Java 都是流行的程序设计语言'
In: 'Hello {0} and {1}! {0}和{1}都是流行的程序设计语言'.format('Python', 'Java')
Out: 'Hello Python and Java! Python 和 Java 都是流行的程序设计语言'
# {}中没有指定序号
In: '{}和{}都是流行的程序设计语言'.format('Python','Java')
Out: 'Python 和 Java 都是流行的程序设计语言'
```

- 使用键值对匹配

如果 format()函数中的参数以键值对形式给出（如下例所示），且替换字段名也用“键”表示，那么它将匹配 format()函数中的对应参数名称（键），并用匹配的同名参数的“值”置换替

换字段而实现格式化输出。

【例】使用键值对匹配。

```
In: '书名:{name}，单价：{price}'.format(name='Python 程序设计', price=50.0)
Out: '书名:Python 程序设计，单价：50.0'
```

- 使用索引匹配

如果 format()函数中的参数是列表或元组，那么可以使用索引进行匹配。关于列表和元组的内容将在第 3 章中介绍。

【例】使用索引匹配字符串。

```
In: book = ['Python 程序设计', 50.0]
In: address = ['广东', '广州']
# 字段名“0[0]”中的第一个 0 表示匹配 format()函数中参数的序号，第二个 0 为参数的索引
In: '书名:{0[0]}，单价：{0[1]}。地点：{1[0]}省 {1[1]}市'.format(book, address)
Out: '书名:Python 程序设计，单价：50.0。地点：广东省 广州市'
```

在格式字符串的替换字段中，还可以设置格式说明标记，标记的语法格式如下：

```
[[fill] align] [sign] [width] [,] [.precision] [type]
```

fill：设置填充的字符，可省略，默认为空格。

align：设置对齐方式，^、<、>分别对应居中、左对齐、右对齐，可省略，默认右对齐。

sign：设置数值型数据前的符号，+表示在正数前加正号，-表示在正数前不变，空格表示在正数前加空格，可省略。

width：设置格式化后的字符串所占宽度，可省略。

逗号（,）：为数字添加千分位分隔符，可省略。

precision：设置数值型数据保留的小数位数，可省略。

type：设置格式化类型符，格式化类型符如表 2.4 所示，可省略。

表 2.4　format()的格式化类型符

符　号	说　明
b	格式化二进制数
c	格式化字符及其 ASCII 码
s	字符串
d	整数
o	无符号八进制数
x	无符号十六进制数（小写）
X	无符号十六进制数（大写）
f	小数，可指定小数点后的精度
e	用科学记数法格式化定点数
g	根据值的大小决定使用 f 或 e
%	浮点数按百分比格式输出

【例】使用格式标记格式化字符串。

```
# 长度为 10，使用二进制数值，居中对齐
In: '{0:^10b}'.format(12)
Out: '   1100   '
# 分别用百分比和科学记数法方式显示，保留小数位 2 位
```

```
In: '{0:.2%}和{1:.2e}'.format(0.1234, 123456)
Out: '12.34%和 1.23e+05'
# 长度为 10，保留 2 位小数，使用千位位分隔符，右对齐，长度不够用“#”填充
In: '{:#>10,.2f}'.format(1234.5678)
Out: '##1,234.57'
```

2.2 变量

2.2.1 关键字和标识符

关键字和标识符是计算机程序语言中的基本语法元素，是编写程序的基础。下面介绍关键字和标识符。

1. 关键字

关键字（也称保留字）是程序语言中被赋予特定含义和特定作用的单词。Python 定义的关键字如表 2.5 所示。

表 2.5 Python 的关键字

and	as	assert	async	await	break
class	continue	def	del	elif	else
except	False	finally	for	from	global
if	import	in	is	lambda	None
nonlocal	not	or	pass	raise	return
True	try	while	with	yield	

在 Python 中，要注意 True、False 和 None 三个关键字的首字母是大写的。可以使用 help() 函数查看 Python 中的关键字。

```
In: help('keywords')
Here is a list of the Python keywords.  Enter any keyword to get more help.
False                 class                 from              or
...  ...  ...
break             for               not
```

2. 标识符

标识符是用来标识程序中的变量、函数、类、模块及其他对象的字符串。虽然程序设计者可以自主设定程序中的标识符，但需要遵循如下规定：

（1）标识符可以包含字母、数字及下画线（_），但必须以一个非数字字符开始。

（2）标识符区分大小写，长度没有限制。

（3）保留关键字不能用作标识符。

下面是 Python 中合法的标识符：

```
student, _num, a1, 姓名
```

下面则是 Python 的非法标识符：

```
2count, a-b, for, my name, high@
```

此外，标识符最好能在一定程度上反映它所标识的变量、函数、对象等的实际意义，尽量符合见名知义的原则，从而增强程序的可读性。

2.2.2　变量

一个程序中通常都会有被处理的对象，而这些对象在被处理之前要以特定的类型存放在内存中，需要时再取出来处理。为了理解和使用方便，要为存放数据的内存位置设定一个名字。用标识符命名的存储单元称为变量，变量用来存储数据，通过标识符可以获取变量的值，也可以对变量进行赋值。

使用赋值语句可以将指定数据存放到指定的变量中。赋值语句的作用就是将一系列数据值存放到一系列变量中。Python 中的赋值运算符号是“=”，其语句格式如下：

```
变量名 = 表达式值
```

=：赋值运算符号。

变量名：被赋值的目标对象。

表达式值：由变量和运算符构成的运算式，计算结果将赋给左边的变量。

赋值语句将“=”右边的值传递给“=”左边的目标对象（变量），这个过程称为变量赋值。

【例】变量赋值。

```
In: age = 18
```

此时，Python 首先会在内存中找到一个足以存放数值 18 的位置，并将此位置的类型设为整型，然后把数值 18 存放到此位置，并以 age 这个名称进行命名。这时，我们可以认为 age 是存储了数值 18 的变量。Python 的变量不需要像编译型语言（如 C 语言）中的变量那样先声明再使用，Python 的变量可以直接赋值，在赋值的同时即声明了变量，并依据所赋予的值自动地确定数据类型。

变量的值在程序运行中是可以改变的。

【例】改变变量的值。

```
In: age = 18
In: print(age)
18

In: age = 20                # 变量值发生改变
In: print(age)
20
```

Python 的变量允许在程序运行过程中动态地改变数据类型，而编译型语言的变量通常不允许这种修改。

【例】改变变量的数据类型。

```
In: age = 18
In: print(age)
18

In: age = '18 岁'           # 改变了变量的数据类型，由整型变为字符串类型
In: print(age)
18 岁
```

需要注意的是，变量名在被引用之前必须先赋值。

【例】变量在使用前需要先赋值。

```
In: num1 = 10
```

```
In: num1
Out:10
In: num2 = num1              # 用变量 num1 给变量 num2 赋值
In: num2
Out:10
In: num1 = 20                # 改变变量 num1 的值
In: num2
Out:10
```

上面的代码在使用变量 num1 给变量 num2 赋值之前，num1 必须先被赋值。此外，虽然第三个赋值语句改变变量 num1 的值为 20，但并没有影响变量 num2 的值，因为这里的变量 num1 和变量 num2 分别对应不同的内存区域。

如果需要给多个变量赋值，那么可以采用下面三种方式：

方式 1：

```
In: a = 10
In: b = 20
In: c = 30
```

方式 2：

```
In: a = 10; b = 20; c = 30
```

方式 3：

```
In: a, b, c = 10, 20, 30
```

在方式 1 中，每个变量的赋值单独占一行。在方式 2 中，将三行赋值写在同一行，此时需要使用分号（;）进行分隔。在方式 3 中，三个变量使用一个赋值运算符（=）进行赋值，此时等号左边的变量用逗号（,）分隔，右边的值也用逗号分隔。

2.3 运算符与表达式

2.3.1 运算符

运算符是 Python 用于表示数据对象执行某种运算的符号。运算符按其功能可以划分为算术运算符、关系运算符、逻辑运算符、赋值运算符、位运算符、成员运算符、身份运算符等。按照操作数的数量，运算符还可分为一元运算符（如~）和二元运算符（如+、*）等。这里按照运算符的功能将 Python 的基本运算符划分为如下几类：

- 算术运算符：-（求负）、+、-、*、/、//、%、**
- 关系运算符：==、!=、>、>=、<、<=
- 逻辑运算符：not、and、or
- 赋值运算符：=、+=、-=、*=、/=、%=、**=、//=
- 位运算符：&、|、^、~、<<、>>
- 成员运算符：in、not in
- 身份运算符：is、is not

1. 算术运算符

算术运算符是执行算术运算的符号，它们是最基本、最常见的运算符。算术运算符如表 2.6 所示。

表 2.6　算术运算符

运　算　符	功　　能	示　　例
-	求负数运算	-a
+	加运算	a + b
-	减运算	a - b
*	乘运算	a * b
/	除运算	a / b
//	整除运算	a // b
%	求余（取模）运算	a % b
**	幂（指数）运算	a ** b

【例】算术运算符的应用。

```
In: x = 13; y = 4
In: -x
Out: -13
In: x + y
Out: 17
In: x * y
Out: 52
In: x / y
Out: 3.25

In: x // y              # 整除运算
Out: 3
In: x % y               # 求余运算
Out: 1
In: x ** y              # 幂（指数）运算
Out: 28561
```

2．关系运算符

关系运算符是用于比较两个数据大小的运算符，其运算的结果为逻辑值 True 或 False。关系运算符都是二元运算符，如表 2.7 所示。

表 2.7　关系运算符

运　算　符	功　　能	示　　例
==	等于	a == b
!=	不等于	a != b
>	大于	a > b
>=	大于等于	a >= b
<	小于	a < b
<=	小于等于	a <= b

【例】关系运算符的应用。

```
In: x = 2; y = 3
In: x == y
Out: False
```

```
In: x != y
Out: True
In: x > y
Out: False
In: x <= y
Out: True
```

3．逻辑运算符

逻辑运算符是操作数和运算结果都是逻辑值的运算符，逻辑运算符如表 2.8 所示。

表 2.8　逻辑运算符

运　算　符	功　　能	示　　例
not	逻辑非	not x
and	逻辑与（x 和 y 必须都为真，结果才为真）	x and y
or	逻辑或（x 或 y 有一个为真，结果就为真）	x or y

【例】逻辑运算符的应用。

```
In: x = True
In: y = False
In: not x
Out: False
In: not y
Out: True
In: x and y
Out: False
In: x or y
Out: True
```

4．赋值运算符

赋值运算符是将一系列值存放到一系列变量中的运算符，“=”是 Python 的基本赋值运算符，其格式为

```
变量名 = 表达式值
```

在基本赋值符“=”前面加上其他运算符，就构成了扩展赋值运算符。例如，“a += 5”等价于“a = a + 5”。扩展运算符先对“=”左边的变量进行其他运算，再将运算的结果赋值给“=”左边的变量。Python 中的扩展赋值运算符如表 2.9 所示。

表 2.9　扩展赋值运算符

运　算　符	功　　能	示　　例
+=	加法赋值运算符	b += a 等效于 b = b + a
-=	减法赋值运算符	b -= a 等效于 b = b - a
*=	乘法赋值运算符	b *= a 等效于 b = b * a
/=	除法赋值运算符	b /= a 等效于 b = b / a
//=	整除赋值运算符	b //= a 等效于 b = b // a
%=	求余赋值运算符	b %= a 等效于 b = b % a
**=	幂赋值运算符	b **= a 等效于 b = b ** a

【例】赋值运算符的应用。

（1）+=运算符

```
In: a = 10
In: b = 20
In: b += a
In: b
Out: 30
```

（2）-=运算符

```
In: a = 10
In: b = 20
In: b -= a
In: b
Out: 10
```

（3）*=运算符

```
In: a = 10
In: b = 20
In: b *= a
In: b
Out: 200
```

5．位运算符

位运算符是将操作数视为二进制数并按位进行运算的运算符。Python 的位运算符如表 2.10 所示。

表 2.10　位运算符

运　算　符	功　能	示　例
~	按位取反运算符	~ a
&	按位与运算符	a & b
|	按位或运算符	a | b
^	按位异或运算符	a ^ b
>>	右移位运算符	a >> b
<<	左移位运算符	a << b

【例】赋值运算符的应用。

（1）~运算符

```
In: a = 10          # 二进制数为 00001010
In: ~a              # ~(00001010)的结果为 11110101，对应的是十进制数-11 的补码
Out: -11            # 所以显示结果为-11
```

（2）&运算符

```
In: a = 10
In: b = 7
In: a & b     # 00001010 & 00000111 的结果为 00000010，对应十进制数 2
Out: 2
```

（3）|运算符

```
In: a = 10
```

```
In: b = 7
In: a | b     # 00001010 | 00000111 的结果为 00001111，对应十进制数 15
Out: 15
```

（4）^运算符

```
In: a = 10
In: b = 7
In: a ^ b     # 00001010 ^ 00000111 的结果为 00001101，对应十进制数 13
Out: 13
```

（5）>>运算符

```
In: a = 11          # 二进制数为 00001011
In: a >> 2          # 右移 2 位变为 00000010，对应十进制数 2
Out: 2
```

（6）<<运算符

```
In: a = 11
In: a << 2          # 左移 2 位变为 00101100，对应十进制数 44
Out: 44
```

6．成员运算符

Python 中的成员运算符是用于测试某个对象是否为某个容器对象中的数据成员的运算符，这里的容器对象是指字符串、列表、元组或集合。Python 的成员运算符如表 2.11 所示。

表 2.11　成员运算符

运　算　符	功　　能	示　　例
in	包含于	a in b
not in	不包含	a not in b

【例】成员运算符的应用。

（1）in 运算符

```
In: s = 'Python'
In: 'P' in s
Out: True
In: 'p' in s
Out: False

In: 3 in [1, 2, 3]
Out:True
```

（2）not in 运算符

```
In: s = 'Python'
In: 'p' not in s
Out: True
```

7．身份运算符

身份运算符是判定两个变量所指的存储区域是否相同的运算符。Python 的身份运算符如表 2.12 所示。

表 2.12　身份运算符

运　算　符	功　　能	示　　例
is	等同	a is b
is not	不等同	a is not b

【例】身份运算符的应用

（1）is 运算符

```
In: a = [1, 2]
In: b = a          # a, b 指向同一内存区域
In: a is b         # a, b 等同
Out: True
In: c = a.copy()   # 通过 a.copy()，a 和 c 指向不同的内存区域
In: a is c
Out: False         # a, c 不等同
```

（2）is not 运算符

```
In: a is not c
Out: True
```

2.3.2　表达式

表达式是将各种数据（包括值、变量和函数）通过运算符按一定规则连接起来的式子。表达式执行指定的运算并返回结果，如“a + b”和“x > y”等都是表达式。表达式的计算结果通常用于对变量赋值或作为程序控制的条件。

表达式的计算需要按照运算符的优先顺序从高到低依次进行。运算符的优先级决定了表达式中各运算符执行的先后顺序；大体上说，算术运算符的优先级高于关系运算符的优先级，关系运算符的优先级高于逻辑运算符的优先级，逻辑运算符的优先级高于赋值运算符。Python 运算符优先级从高到低的顺序如表 2.13 所示。

表 2.13　运算符的优先级（从高到低）

运　算　符	描　　述
**	指数（最高优先级）
~ , -	按位翻转，求负数
*, /, %, //	乘、除、取模和整除
+, -	加法、减法
>>, <<	右移、左移运算符
&	按位与运算符
^, |	按位异或、或运算符
<=, <, >, >=	比较运算符
==, !=	等于/不等于运算符
=, +=, %=, -=, *=, /=, //=, **=	赋值运算符
is, is not	身份运算符
in, not in	成员运算符
not	逻辑非运算符
and	逻辑与运算符
or	逻辑或运算符
lambda	Lambda 表达式

表达式可以使用括号“()”显式地改变括号内外运算符的优先顺序，使得括号内的运算优先于括号外的运算。此外，适当地使用括号也可使表达式的结构更清晰。

【例】

```
not a > b and c + d or e == f
```

可以使用括号显式地写成

```
(not (a > b) and (c + d))  or  (e == f)
```

这样就更清楚地表明了各运算的顺序，使程序的可读性更强。

2.3.3 数据类型转换

Python 规定表达式中各项数据的类型必须一致。如果数据类型不一致，那么就要对数据进行类型转换才能计算表达式的值。数据类型转换可分为自动转换和显式转换两种情况。

1．自动转换

如果表达式中出现的数据类型不一致，那么 Python 会检查这些数据是否可以转换为该表达式需要的类型。如果可以，那么将原数据类型转换为表达式所需要的类型；如果不可以，那么报告类型错误。这种数据类型转换称为自动转换。例如，表达式“1.2 + 3”中的两个操作数类型不同，Python 会将整型数 3 转换为浮点数 3.0，然后执行表达式“1.2 + 3.0”的运算。

在数值型数据类型中，Python 自动转换的规则如下：

- 整型数据可以自动在数据后面加上“.0”转换为浮点型数据。
- 非复数型数据可以自动在数据后面加上“0j”转换为复数型数据。

2．显式转换

Python 可以将整型数据自动转换为浮点型数据，但不可以将浮点型数据自动转换为整型数据。如果需要将浮点型数据转换为整型数据，那么要使用 Python 提供的内置函数进行转换，这种转换称为显式转换。表 2.14 中给出了 Python 提供的常用类型转换函数。

表 2.14 Python 的常用类型转换函数

函数格式	描　述
int(x [,base])	将字符串或其他数值类型的数据转换为整型
float(x)	将字符串或其他数值类型的数据转换为浮点数
complex(real,imag)	转换为复数，real 可以是字符串或数值，imag 只能为数字类型（默认为 0）
str(x)	将数值转换为字符串
repr(x)	返回一个对象的字符串格式
eval(str)	将 str 转换为表达式并返回表达式的运算结果
tuple(seq)	将 seq 转换为元组，seq 可为列表、集合或字典等
list(seq)	将 seq 转换为列表，seq 可为元组、集合或字典等
set(seq)	将一个可迭代对象 seq 转换为集合
chr(x)	返回整数 x 对应的 unicode 字符
ord(ch)	返回字符 ch 对应的 ASCII 码或 unicode 码
hex(x)	将整数 x 转换为十六进制字符串
oct(x)	将整数 x 转换为八进制字符串

【例】数据类型转换的应用。

（1）int 转换函数

```
In: int(12.6)      # 将数值转为整型，截掉小数部分，只保留整数部分
Out: 12
In: ord('A')       # 将字符'A'转换为 ASCII 编码
Out: 65
In: ord('a')       # 将字符'a'转换为 ASCII 编码
Out: 97
```

（2）转换为 complex 型

```
In: a = 1; b = 2.3
In: complex(a, b)
Out: (1+2.3j)
```

（3）字符串转换为数值

```
In: y = eval('1.2+3.5')     # eval 函数可视为去掉字符串两侧的定界符，计算内部的表达式
In: y
Out: 4.7
In: float('3.5')            # 字符串转换为浮点数
Out:3.5
In: int('3')                # 字符串转换为整数
Out:3
In: int('3.5')              # 将报错，字符串'3.5'不能转换为整数
ValueError: invalid literal for int() with base 10: '3.5'
```

2.4　输入与输出

2.4.1　输入函数 input()

如果程序运行过程中要从键盘获取数据，那么就要使用 Python 的内置函数 input()。值得注意的是，不论用户输入的是什么数据，函数 input()一律将接收到的数据视为字符串。函数 input()的语法格式如下：

```
input(<prompt>)
```

<prompt>：字符串，用于提示用户输入的文本内容。

Python 执行 input()函数时，会将<prompt>中的文本内容输出显示，作用是提示用户需要输入数据。用户输入数据并按 Enter 键后，input()函数会以字符串的形式获取用户输入的任何数据。

【例】从键盘输入数据。

```
In: num = input('请输入一个数值：')
请输入一个数值：10
In: num
Out: '10'
In: num + 20       # 将报错，字符串不能和整数直接相加
TypeError: can only concatenate str (not "int") to str

In: int(num) + 20  # 将字符串转为整数再执行算术运算，也可写为 eval(num)+20
Out: 30
```

系统执行 input()函数时会显示 input()函数的提示内容（“请输入一个数值：”），然后光标停留在提示内容后面等待用户输入数据。用户输入数据 10 并按 Enter 键后，input()函数读取的数据是字符串'10'而不是整数 10。赋值运算符将字符串'10'赋值给变量 num。注意字符串不能直接和数值做算术运算。

2.4.2 输出函数 print()

通常情况下，需要将程序运行处理后的结果输出显示。Python 的内置函数 print()可以显示输出其参数指定的内容。print()函数的语法格式如下：

```
print(<expr>[,<expr>,…]  [,sep=<string>] [,end=<string>])
```

<expr>：输出内容的表达式，可省略。

sep：多个输出数据之间的分隔符，可省略，默认值是单个空格' '。

end：末尾输出内容，可省略，默认值是换行符'\n'。

print()函数会依次显示输出各个<expr>表达式的结果。print()函数支持一个或多个表达式运算结果的输出，表达式之间要用逗号“,”分隔。如果有多个表达式的值要输出，那么可以通过设置 sep 参数设定输出内容之间的分隔符。通常情况下一个 print()函数输出一行内容，也就是说 print()函数在输出内容后会自动换行。通过设置 end 参数可以设定 print()函数输出后不换行。

【例】输出显示。

（1）输出多个数据

```
In: a, b, c = 10, 20, 30
In: print(a, b, c)
10 20 30                                # 显示的数据间默认间隔一个空格
```

（2）输出多个数据，以指定的字符连接

```
In: a, b, c = 10, 20, 30
In: print(a, b, c, sep='**')            # sep 参数指定数据间的间隔符
10**20**30
```

（3）输出多行数据，每个 print()函数占一行

```
In: a, b, c = 10, 20, 30
In: print(a); print(b); print(c)        # print()函数默认换行，因此分三行输出
10
20
30
```

（4）多个 print()函数输出，每个 print()函数输出后都不换行

```
In: a, b, c = 10, 20, 30
In: print(a, end=''); print(b, end=''); print(c, end='') # 设置 end 参数为空，不换行
102030
```

2.5 内置函数

计算机程序中的函数是用于实现某一特定功能的程序语句集合。一个函数通常实现较为单一的功能，且具有相对的独立性。Python 提供了各种功能的内置函数供用户使用，这些内置函数会随 Python 启动而自动加载。2.1.4 节介绍了有关类型转换的 Python 内置函数，本节介绍 Python 内置的数学运算函数、字符串处理函数和其他函数。

2.5.1　数学运算函数

Python 内置的常用数学运算函数如表 2.15 所示。

表 2.15　常用的数学运算函数

函　数　名	功　　能
abs(x)	返回 x 的绝对值
divmod(x,y)	返回一个(x//y, x%y)的元组，即 x 除以 y 的商和余数
max(seq)	返回 seq 序列中的最大值
min(seq)	返回 seq 序列中的最小值
pow(x,y)	返回 x 的 y 次方
round(x, precision)	返回浮点数 x 的四舍五入值
sum(seq)	返回 seq 序列元素的和

【例】数学运算函数的应用。

```
In: abs(-1.2)                # 绝对值
Out: 1.2

In: divmod(16, 5)            # 返回整除的商和余数
Out: (3, 1)

In: max(2, -1, 8, 3)         # 最大值
Out: 8

In: min(2, -1, 8, 3)         # 最小值
Out: -1

In: pow(2, 3)                # 幂函数
Out: 8

In: round(3.14159)           # 默认四舍五入取整
Out: 3
In: round(3.14159, 3)        # 四舍五入，小数点后保留 3 位
Out: 3.142

In: sum([1, 2, 3, 4, 5])     # 求和
Out: 15
```

2.5.2　字符串处理函数

针对字符串的字母大小写转换、字符的查找与替换、统计、对齐及拆分合并等操作，Python 提供了如表 2.16 至表 2.21 所示的内置函数。

表 2.16 常用的字符串转换函数

函 数 名	功 能
capitalize()	字符串首字母大写，其余字母小写
lower()	字符串中所有字母都小写
swapcase()	字符串中所有字母大小写互换
title()	字符串中每个单词的首字母大写，其余字母小写
upper()	字符串中所有字母都大写

【例】字符串转换函数的应用。

```
In: 'pYthon'.capitalize()
Out: 'Python'

In: 'PyTHon'.lower()
Out: 'python'

In: 'python'.upper()
Out: 'PYTHON'

In: 'Python'.swapcase()
Out: 'pYTHON'

In: 'hello python'.title()
Out: 'Hello Python'
```

表 2.17 常用的字符串搜索替换函数

函 数 名	功 能
find(substr, [start,[end]])	返回字符串中第一次出现 substr 的位置，若不包含 substr 则返回-1
index(substr, [start,[end]])	与 find()相同，但字符串中无 substr 时返回一个运行时错误
replace(oldstr, newstr,[count])	返回将字符串中 oldstr 串替换为 newstr 串后的新字符串，count 为替换次数
rfind(substr, [start,[end]])	返回从右侧开始查询字符串中第一次出现 substr 的位置，若无 substr 则返回-1
rindex(substr, [start,[end]])	与 rfind()相同，但字符串中无 substr 时返回一个运行时错误

【例】字符串搜索替换函数的应用。

```
In: 'Hello,Python!'.find('o')
Out: 4
In: 'Hello,Python!'.find('o', 6)
Out: 10
In: 'Hello,Python!'.index('th')
Out: 8
In: 'Hello,Python!'.find('th')
Out: 8
In: 'Hello,Python!'.replace('Python', 'Java')
Out: 'Hello,Java!'
In: 'Hello,Python!'.rfind('o')
Out: 10
```

表 2.18　常用的字符串统计函数

函 数 名	功　能
count(substr, [start,[end]])	在[start, end)位置范围内查询 substr 在字符串中出现的次数，若不出现则返回 0
len(str)	返回字符串 str 的长度

【例】字符串统计函数的应用。

```
In: 'Hello,Python!'.count('o')        # 统计'o'出现的次数
Out: 2
In: 'Hello,Python!'.count('o', 0, 8) # 在[0,8)位置统计'o'出现的次数
Out: 1
In: 'abab'.count('o')                 # 统计'o'出现的次数，不出现则返回 0
Out: 0
In: len('Hello,Python!')              # 字符串长度
Out: 13
```

表 2.19　常用的字符串判断函数

函 数 名	功　能
startswith(substr)	判断字符串是否以 substr 开头，若是则返回 True，否则返回 False
endswith(substr)	判断字符串是否以 substr 结尾，若是则返回 True，否则返回 False
isalpha()	判断字符串是否只由字母组成，若是则返回 True，否则返回 False
isdigit()	判断字符串是否只由数字组成，若是则返回 True，否则返回 False
islower()	判断字符串中的字母是否全为小写，若是则返回 True，否则返回 False
isnumeric()	判断字符串中是否只含数字字符，若是则返回 True，否则返回 False
isspace()	判断字符串中是否只含空格，若是则返回 True，否则返回 False
isupper()	判断字符串中的字母是否全是大写，若是则返回 True，否则返回 False
isalnum()	判断字符串是否全由字母或数字组成，若是则返回 True，否则返回 False

【例】字符串判断函数的应用。

```
In: 'Hello,Python'.endswith('Python')
Out: True
In: 'Hello,Python'.startswith('Python')
Out: False
In: 'Python3'.isalnum()
Out: True
In: '123'.isdigit()
Out: True

In: '123'.isnumeric()
Out: True
In: 'hello,python!'.islower()
Out: True
In: 'Hello,Python!'.isupper()
Out: False
```

表 2.20 常用的字符串拆分合并函数

函 数 名	功 能
join(seq)	将 seq 中的所有元素以指定的字符连接，返回连接得到的字符串
split(sep,[num])	以 sep 为分隔符（默认空格），对字符串拆分（num 为拆分次数），返回拆分得到的列表

【例】字符串拆分合并函数的应用。

```
In: lst = ['Hello', 'Python']
In: ','.join(lst)
Out: 'Hello,Python'
In: '+'.join(['a', 'b', 'c', 'd', 'e'])
Out: 'a+b+c+d+e'

In: 'Hello,Python'.split(',')    # 以逗号为分隔符
Out: ['Hello', 'Python']
In: lst = 'Python is a programing language'.split()
                                  # 默认以空格为分隔符拆分字符串，返回列表
In: lst
Out:['Python', 'is', 'a', 'programing', 'language']
In: '-'.join(lst)
Out: 'Python-is-a-programing-language'
```

表 2.21 常用的字符串对齐函数

函 数 名	功 能
center(width [,fillchar])	返回长度为 width 的字符串，原串居中对齐，长度不足部分填充 fillchar（默认空格）
ljust(width [,fillchar])	返回长度为 width 的字符串，原串左对齐，长度不足部分填充 fillchar
rjust(width [,fillchar])	返回长度为 width 的字符串，原串右对齐，长度不足部分填充 fillchar
zfill(width)	返回长度为 width 的字符串，原串右对齐，长度不足部分填充 0

【例】字符串对齐函数的应用。

```
In: 'Python'.center(10, '*')
Out: '**Python**'
In: 'Python'.ljust(10)
Out: 'Python    '       # 默认填充空格

In: 'Python'.rjust(10, '*')
Out: '****Python'
In: 'Python'.zfill(10)
Out: '0000Python'
```

2.5.3 其他函数

Python 还提供了 help()、dir()、type()和 id()等内置函数，这些函数可用于查询对象或方法的相关信息，它们的具体功能如表 2.22 所示。

表 2.22　其他常用的函数

函　数　名	功　　能
dir([object])	不带 object 时，返回当前范围内的变量、方法和定义的对象列表；带 object 时，返回 object 的属性、方法列表
help([object])	返回指定 object（模块、类型、对象、方法、属性）的帮助信息
id(object)	返回 object 对象的唯一标识符（内存地址）
type(object)	返回 object 对象的数据类型

【例】help()、type()和 id()函数的应用。

（1）help()函数

```
In: help(input)
Help on built-in function input in module builtins:

input(prompt=None, /)
    Read a string from standard input. The trailing newline is stripped.
    The prompt string, if given, is printed to standard output without a
    trailing newline before reading input.
```

（2）type()函数

```
In: x = 123
In: type(x)        # 返回 x 变量的数据类型
Out: int
In: type(3.14), type(True), type([1,2,3])
Out: (float, bool, list)
```

（3）id()函数

```
In: x = 123
In: id(x)          # 返回 x 变量的唯一标识符，类似内存地址
Out: 1364906032
In: x = 'abc'      # 赋予新值
In: id(x)          # 可见 x 变量的内存地址已变化
Out: 1454907035    # 说明前后两个 x 变量只是变量名相同，但内存指向不同
```

2.6　常用模块

Python 将有组织的程序代码片段称为模块（Module）。模块包含了 Python 对象的定义和语句，并以 Python 文件（扩展名为“.py”）保存在外存中。用户可以使用 Python 标准库中的模块，也可以使用第三方模块。

Python 标准库（也称内置库或内置模块）是 Python 的重要组成部分，它会随 Python 解释器一起安装到系统中。Python 标准库中提供了很多模块，下面介绍其中一些常用的模块。

2.6.1　模块导入

编程时如果要用到 Python 内置模块，那么就要使用 import 语句将该模块导入。Python 将模块导入程序的方式有以下几种。

（1）常规导入

常规导入就是将整个模块直接导入，被导入模块的所有变量、对象和函数在编程时都可以使

用。这种导入的语法格式如下：

```
import 模块名
```

模块名：需要导入的模块名称。

【例】导入 math 数学模块。

```
In: import math                          # 导入整个 math 模块
In: math.pi                              # 调用 math 模块中的 pi 常量
Out: 3.141592653589793
In: math.sin(math.pi / 2)                # 调用 math 模块中的 pi 常量和 sin()函数
Out: 1.0
```

上面的代码中使用“import math”语句导入整个 math 数学模块，因此后面的代码可以使用该模块的所有变量、对象和函数。

需要注意的是，采用这种方式导入模块后，必须采用“模块名.变量名/对象名/函数名”的方式使用模块中的变量、对象或函数，如上面代码中的 math.pi 和 math.sin()。

（2）使用别名

在导入模块时可以给模块取简短的别名，使用模块时用别名代替模块名，其语法格式如下：

```
import 模块名 as 别名
```

模块名：需要导入的模块名称。

别名：给模块取的别名。

【例】使用别名导入模块。

```
In: import math as m                     # 导入 math 模块并取别名 m
In: m.pi                                 # 调用 m 模块中的 pi 常量
Out: 3.141592653589793
In: m.sin(m.pi / 2)                      # 调用 m 模块中的 pi 常量和 sin()函数
Out: 1.0
```

上面的代码中使用“import math as m”语句导入 math 模块并给它取一个别名 m，因此使用 math 模块中的常量 pi 和 sin()函数时就可以用 m.pi 和 m.sin()的方式。

（3）部分导入

如果不导入整个模块而只导入模块的某些变量、函数或子模块，那么可采用部分导入的方式导入指定的内容，其语法格式如下：

```
from 模块名 import 变量名/函数名/子模块 as 别名
```

模块名：需要导入的模块名称。

变量名/函数名/子模块：指定导入的变量名、函数名或子模块，有多个时用逗号（,）隔开。

别名：给导入的变量名或函数名取别名，可省略。

【例】部分导入的应用。

```
In: from math import pi, sin             # 只导入 math 模块的常量 pi 和 sin()函数
In: pi                                   # 调用时直接写常量名 pi
Out: 3.141592653589793
In: sin(pi / 2)                          # 调用时直接写常量名 pi 和 sin()函数名
Out: 1.0
```

上面的代码中使用“from math import pi, sin”语句只导入 math 模块中的 pi 常量和 sin()函数。于是调用它们时不再需要在常量 pi 和 sin()函数前面加上模块名，而只需直接给出常量 pi 和 sin()函数名。此外，如果给导入的变量名或函数名取了别名，那么调用它们时就可以使用它们的别名。

【例】对导入的变量名、对象名或函数名取别名。

```
In: from math import pi as pai, sin as fsin    # 分别给常量 pi 和 sin()函数取别名
In: pai                                        # 调用时用常量 pi 的别名
Out: 3.141592653589793
In: fsin(pai / 2)                              # 调用时用常量 pi 和 sin()函数名的别名
Out: 1.0
```

（4）全部导入

如果要导入模块的所有变量、对象或函数，那么采用全部导入方法，其语法格式如下：

```
from 模块名 import  *
```

模块名：需要导入的模块名称。

*：模块内的所有成员。

【例】全部导入。

```
In: from math import *                  # 将 math 模块的所有成员都导入
In: pi                                  # 调用时直接写常量名 pi
Out: 3.141592653589793
In: sin(pi / 2)                         # 调用时直接写常量名 pi 和 sin()函数名
Out: 1.0
```

上面的代码中使用“from math import *”语句导入了 math 模块的所有变量、对象或函数，后面的代码可以直接使用 math 模块的所有成员。例如，使用模块中的 pi 常量和 sin()函数时，不需要在它们前面再添加“math.”。如果采用这种方式导入多个模块，那么可能无法区分调用的函数究竟是哪个模块中的，所以 Python 不建议使用这种导入方式，一般多采用第一种或第二种导入方式。

2.6.2　math 数学模块

math 模块提供了 4 个数学常量和 44 个函数。要使用 math 模块，首先需要使用 import 语句导入模块，语句如下：

```
import  math
```

表 2.23 给出了 math 模块中的常用常量和函数，其他常量与函数可以使用 dir(math)命令或 Python 帮助文档查阅。

表 2.23　math 模块中的常用常量和函数

函 数 名	功　能
math.e	自然常数 e
math.pi	圆周率 pi
math.acos(x)	返回 x 的反三角余弦值
math.asin(x)	返回 x 的反三角正弦值
math.atan(x)	返回 x 的反三角正切值
math.ceil(x)	返回>=x 的最小整数
math.cos(x)	返回 x（弧度）的三角余弦值
math.copysign(x, y)	若 y<0，则返回-1 乘以 x 的绝对值，否则返回 x 的绝对值
math.degrees(x)	弧度转换为角度
math.exp(x)	返回 e 的 x 次方
math.fabs(x)	返回 x 的绝对值
math.factorial(x)	返回 x 的阶乘
math.fmod(x, y)	返回 x%y（取余）

（续表）

函 数 名	功 能
math.floor(x)	返回<=x 的最大整数
math.isinf(x)	若 x 为无穷大则返回 True，否则返回 False
math.isnan(x)	若 x 不是数字则返回 True，否则返回 False
math.log(x[, base])	返回 x 的以 base 为底的对数，base 默认为 e
math.log10(x)	返回 x 的以 10 为底的对数
math.modf(x)	返回 x 的小数和整数
math.pow(x, y)	返回 x 的 y 次方
math.radians(x)	角度转换为弧度
math.sin(x)	返回 x（弧度）的三角正弦值
math.sqrt(x)	返回 x 的平方根
math.tan(x)	返回 x（弧度）的三角正切值
math.trunc(x)	返回 x 的整数部分

【例】math 模块的常量和函数的应用。

```
In: import math
In: math.e
Out: 2.718281828459045
In: math.pi
Out: 3.141592653589793
In: math.floor(3.14)                    # 返回<=3.14 的最大整数
Out: 3
In: math.modf(3.14)                     # 分别返回小数和整数部分
Out: (0.14000000000000012, 3.0)
In: math.radians(45)                    # 角度转换为弧度
Out: 0.7853981633974483
In: math.fabs(-1.23)                    # 绝对值
Out: 1.23
In: math.fmod(8, 3)                     # 求余数
Out: 2.0
In: math.exp(2)                         # e 的 2 次方
Out: 7.38905609893065
In: math.pow(2, 3)                      # 求幂（指数）
Out: 8.0
In: math.factorial(5)                   # 阶乘函数
Out: 120
In: math.sin(math.pi / 2)               # 正弦函数
Out: 1.0
```

2.6.3 random 随机数模块

Python 的 random 模块是用于生成随机数的模块。random 模块提供了生成不同类型随机数的函数，但所有这些函数都是基于 random.random()函数扩展实现的。使用 random 模块前要先导入该模块，语句如下：

```
import  random
```

表 2.24 中给出了 random 模块中常用的随机数函数。

表 2.24　random 模块中常用的随机数函数

函 数 名	功　能
choice(seq)	返回一个从 seq 中随机选取的元素
randrange([start,] stop [,step])	返回一个在[start, stop)范围内以 step 为步长的整数序列中的随机整数
random()	返回一个在[0, 1)内的随机浮点数
seed()	改变随机数生成器的种子，无返回值
shuffle(seq)	将序列 seq 中的元素顺序打乱，无返回值
randint(min, max)	返回一个[min, max]范围内的随机整数
uniform(min, max)	返回一个[min, max)范围内的随机浮点数
sample(seq, num)	返回从 seq 中随机选取 num 个元素的子序列

【例】常用随机数函数的应用。

```
In: import random
In: random.random()
Out: 0.9551421885236997
In: random.uniform(1, 10)                # 获得[1,10)内的随机浮点数
Out: 9.09577665686459
In: random.randint(1, 10)                # 获得[1,10]内的随机整数，可包含 10
Out: 3
In: random.randrange(3)                  # 获得[0,3)内的一个随机整数，不包含 3
Out: 2
In: random.choice((1, 2, 3, 4, 5))       # 在(1,2,3,4,5)中随机选择一个数
Out: 4
In: random.sample((1, 2, 3, 4, 5), 2)    # 在(1,2,3,4,5)中随机选择两个数
Out: [4, 2]
```

2.6.4　datetime 和 time 模块

Python 的 datetime 模块和 time 模块提供了一些处理日期和时间的函数。time 模块提供了各种关于时间操作的函数。datetime 模块基于 time 模块进行了封装，提供了更多实用的对象和函数。

1. time 模块

time 模块主要包含各种提供日期、时间功能的类和函数。该模块既提供了把日期、时间格式化为字符串的功能，又提供了将字符串转换为日期、时间格式的功能。

time 模块中表示时间的格式主要有三种。

（1）时间戳

时间戳表示从 1970 年 1 月 1 日 00:00:00 开始到现在按秒计算的总秒数。这种时间表示方式广泛应用在 Unix 系统和类 Unix 系统中。

（2）时间元组

Python 在 time 模块内提供了一个 time.struct_time 类，该类代表一个时间对象，主要包含 9 个属性：tm_year、tm_mon、tm_mday、tm_hour、tm_min、tm_sec、tm_wday、tm_yday、tm_isdst，分别表示年、月、日、时、分、秒、周、一年内的第几天和夏令时。

```
In:import time
In:time.localtime()
```

```
Out:time.struct_time(tm_year=2019,tm_mon=9,tm_mday=19,tm_hour=7,tm_min=41,
tm_sec=10, tm_wday=3, tm_yday=262, tm_isdst=0)
```

（3）格式化时间

格式化时间是由字母和数字表示的时间字符串，比如“Sat May 4 15:11:24 2019”。格式化的结构增强了时间字符串的可读性。时间字符串支持的格式符号如表 2.25 所示。

表 2.25 时间字符串支持的格式符号（区分大小写）

格 式	含 义
%a	本地简化星期名称
%A	本地完整星期名称
%b	本地简化月份名称
%B	本地完整月份名称
%c	本地相应的日期和时间表示
%d	一个月中的第几天（01～31）
%H	一天中的第几个小时（24 小时制，00～23）
%I	一天中的第几个小时（12 小时制，01～12）
%j	一年中的第几天（001～366）
%m	月份（01～12）
%M	分钟数（00～59）
%p	本地 am 或 pm 的相应符号
%S	秒（00～61）
%w	一个星期中的第几天（1～7，7 是星期天）
%x	本地相应日期
%X	本地相应时间
%y	去掉世纪的年份（00～99）
%Y	完整的年份
%Z	时区的名字（若不存在则为空字符）

时间字符串示例如下：

```
In: t = time.localtime()                          # 取得当前日期和时间
In: time.strftime('%Y-%m-%d %H:%M:%S', t)         # strftime 将时间变量 t 转换为字符串
Out: '2019-09-19 07:49:12'
In: time.strftime('%x  %X  %B  %A', t)
Out: '09/19/19  07:49:12  September  Thursday'

In: time.strptime('2019-05-01', '%Y-%m-%d')       # strptime 将字符串转换为时间
Out: time.struct_time(tm_year=2019, tm_mon=5, tm_mday=1, tm_hour=0,
     tm_min=0, tm_sec=0, tm_wday=2, tm_yday=121, tm_isdst=-1)
In: help(time.strftime)                           # 显示 strftime 帮助信息
```

2．datatime 模块

datatime 模块重新封装了 time 模块，它以类的方式提供了多种日期和时间的表达方式，提供的类主要有 datetime.date、datetime.time、datetime.datetime、datetime.timedelta 和

datetime.tzinfo。要使用 datetime 模块，首先要导入该模块，语句如下：

```
import datetime
```

下面主要介绍 datetime.datetime 类的使用。

（1）datetime.datetime 类

【例】创建 datetime.datetime 对象。

```
In: import datetime
In: d = datetime.datetime(2019, 5, 1)
In: from  datetime  import  datetime      # 从 datetime 库直接导入 datetime 类
In: datetime(2019, 5, 1)                  # 创建一个指定年、月、日的对象
Out: datetime.datetime(2019, 5, 1, 0, 0)
In: datetime(2019, 5, 1, 14, 32, 45)      # 创建一个指定年、月、日、时、分、秒的对象
Out: datetime.datetime(2019, 5, 1, 14, 32, 45)
```

（2）datetime.datetime 类的方法和属性

为了便于 datetime.datetime 类的使用，Python 提供了多个属性和函数，如表 2.26 所示。

表 2.26　datetime 类的常用函数与属性

方法名/属性名	描　　述
datetime.max	最大时间
datetime.min	最小时间
datetime.today()	返回当前本地时间
datetime.now([tz])	返回当前本地时间，若指定时区 tz，则返回 tz 时区当地时间
datetime.utcnow()	返回当前 UTC 时间
datetime.combine(date,time)	把指定的 date 和 time 对象整合为 datetime 对象
datetime.strptime(date_string,format)	将字符串转换为 datetime 对象
datetime.strftime(dt, format)	将 datetime 对象转为字符串

【例】datetime.datetime 类的常用函数与属性的应用。

```
In: from  datetime  import  datetime
In: datetime.max                                  # 获得最大时间
Out: datetime.datetime(9999, 12, 31, 23, 59, 59, 999999)
In: datetime.today()                              # 获得当前本地时间
Out: datetime.datetime(2019, 9, 21, 14, 24, 43, 283867)
In: datetime.utcnow()                             # 获得当前 UTC（格林尼治标准）时间
Out: datetime.datetime(2019, 9, 21, 6, 24, 52, 758811)

In: dt = datetime.strptime('2019-3-5', '%Y-%m-%d') # 字符串转换为日期对象
In: dt
Out: datetime.datetime(2019, 3, 5, 0, 0)
In: datetime.strftime(dt, '%m-%d')                # 日期转换为字符串
Out: '03-05'
```

（3）datetime.datetime 类的对象的常用方法与属性

除 datetime.datetime 类提供类方法和属性外，datetime.datetime 类的对象也提供一系列属性和方法，其常用的方法与属性如表 2.27 所示（dt 为 datetime.datetime 类的对象）。

表 2.27 datetime.datetime 类的对象的常用方法与属性

方法名/属性名	描　述
dt.year	年
dt.month	月
dt.day	日
dt.hour	时
dt.minute	分
dt.second	秒
dt.microsecond	微秒
dt.tzinfo	时区信息
dt.date()	获取 dt 的 date 对象
dt.time()	获取 dt 的 time 对象
dt.replace()	指定参数替代（年，月，日，时，分，秒，微秒，时区），返回生成的新对象
dt.timestamp()	返回 dt 对应的时间戳
dt.weekday()	返回日期是星期几[0, 6]，0 表示星期一，6 表示星期日
dt.isoformat()	返回格式如'YYYY-MM-DDTHH:MM:SS'的字符串
dt.strftime(format)	返回指定格式的时间字符串，格式符见表 2.25

【例】datetime.datetime 类的对象的常用方法与属性的应用。

```
In: dt = datetime.datetime(2019, 5, 1, 14, 32, 45)
In: dt.year                        # 获得年份
Out: 2019
In: dt.month                       # 获得月份
Out: 5
In: dt.day                         # 获得日
Out: 1
In: dt.date()                      # 获得日期
Out: datetime.date(2019, 5, 1)
In: dt.time()
Out: datetime.time(14, 32, 45)     # 获得时间
In: dt.timestamp()                 # 获得时间戳
Out: 1556692365.0                  # 从 1970 年 1 月 1 日 00:00:00 开始到现在的总秒数
In: dt.weekday()                   # 星期几
Out: 2                             # 星期三
In: dt.isoformat()                 # 按 iso 格式输出日期和时间
Out: '2019-05-01T14:32:45'
In: dt.strftime("%a")              # 输出本地简化星期名称
Out: 'Wed'
In: dt.strftime("%A")              # 输出本地完整星期名称
Out: 'Wednesday'
```

2.7 本章小结

1. Python 提供了丰富的数据类型，包括数值型、字符串、列表、元组、字典等。数值类型包含整型（int）、浮点型（float）、复数型（complex）和布尔型（bool）等，字符串（str）

是最常用的数据类型之一。

2. 标识符是用来表示变量、函数、类、模块及其他对象的名字。标识符由字母、数字或下画线组成。但要注意，标识符中不能有空格，不能以数字开头，也不能是 Python 的关键字。此外，Python 标识符区分大小写。
3. 数据类型的转换有两种形式：自动转换和显式转换。
4. Python 的字符串有两种格式化方法：%和 format。
5. Python 提供了多种类型的运算符。对表达式进行运算时，需要按照运算符的优先顺序从高到低进行，可以使用括号“()”显式地标明运算顺序。
6. 在 Python 中，读取键盘输入使用 input()函数，输出显示使用 print()函数。
7. 内置函数是可自动加载并直接使用的函数，Python 提供很多实现各种功能的内置函数。
8. Python 标准库是 Python 自带的开发包，是 Python 的组成部分，它会随 Python 解释器一起安装到系统中。Python 标准库中提供了很多模块。

习题

一、单项选择题

1. 下面（　　）不是正确的变量名。
   A. num#1　　B. _count　　C. student1　　D. score
2. Python 用字符（　　）表示转义字符。
   A. *　　B. #　　C. \　　D. %
3. 下列选项中，不是 Python 关键字的是（　　）。
   A. elif　　B. class　　C. static　　D. pass
4. 下列运算符中，整除运算符是（　　）。
   A. *　　B. **　　C. /　　D. //
5. 下列运算符中，幂运算符是（　　）。
   A. *　　B. **　　C. /　　D. //
6. 下列运算符中，不等于运算符是（　　）。
   A. !=　　B. =!　　C. <>　　D. ><
7. 下列运算符中，逻辑与运算符是（　　）。
   A. not　　B. and　　C. or　　D. xor
8. 下列运算符中，按位与运算符是（　　）。
   A. *　　B. &　　C. ^　　D. |
9. 下列运算符中，优先级最高的是（　　）。
   A. *　　B. &　　C. >=　　D. not
10. 下面（　　）不是字符串的正确表述。
    A. 'python'　　B. '''python'''　　C. 'py"th"on'　　D. [python]
11. 代码“'%6.2f' % (3.14159)”的运算结果是（　　）。
    A. '3.14159'　　B. '3.14'　　C. ' 3.14'　　D. '3.14 '
12. 代码“'单价:{1}，数量:{0}'.format(10, 20)”的运算结果是（　　）。
    A. '单价:10，数量:20'　　B. '单价:10，数量:10'
    C. '单价:20，数量:10'　　D. '单价:20，数量:20'
13. 代码“'{:#<8}'.format(3.14)”的运算结果是（　　）。
    A. '3.14'　　B. '3.14####'　　C. '####3.14'　　D. '##3.14##'

14. 代码“print('a'.rjust(5, '*'))”的输出结果是（　　）。

A. ****a　　B. a****　　C. **a**　　D. aaaaa

15. 代码“max('hello, Python')”的运算结果是（　　）。

A. 'h'　　B. 'n'　　C. 'y'　　D. ','

16. 下列（　　）函数不属于字符串对齐函数。

A. center()　　B. rjust()　　C. zfill()　　D. fill()

17. 下列（　　）不属于 datetime 模块提供的类。

A. date　　B. time　　C. calendar　　D. tzinfo

18. 下列（　　）函数返回 x 的整数部分。

A. math.ceil(x)　　B. math.fabs(x)

C. math.modf(x)　　D. math.trunc(x)

二、计算题

1．写出下列式子的 Python 表达式。

（1）$2a^2b^5$　　（2）$2a(b+5)^3$　　（3）$\sqrt{b^2-4ac}$　　（4）$\cos(\sin x+1)$

2．计算下列表达式的结果。

（1） 3 + 2**4 - 2　　（2） 7 | 9 & ~2

（3） 8 + 3 * 6 // 2　　（4） 1 + 2**3 + 4 % 5 + (6 // 7) + 8 >= 9

（5） not 1 < 2 > 5 + 4　　（6） x > y and y > z or not x　假定（x=7;y=5; z=3）

（7） 6 * 13 / 2 + 8 - 345 % 10　　（8） 7 * (13 // (4 - 2) - 352) % 10

（9） a > b and c <= d or 2 * a > c　　假定（a=2; b=3; c=4; d=5）

（10） 3 > a * b or a == c and b != c or c > d　假定（a=2; b=3; c=4; d=5）

三、简答题

1. 简述 Python 标识符的命名规则。
2. 数值类型包含哪些类型？
3. 字符串有哪些表示形式？
4. Python 字符串格式化有哪两种方式？
5. 简述 Python 运算符的类型。
6. 列出内置函数中常用的数学运算函数。
7. datetime 模块提供了哪些类？

第 3 章　Python 容器数据类型

除整型、浮点型和布尔型等基本类型外，Python 还提供列表、元组、字典和集合等容器数据类型，容器数据可以容纳批量数据。这些容器数据类型又可分为序列类型、映射类型和集合类型。序列类型包括字符串、列表和元组等，它们的元素都是有序排列的，可以通过索引下标访问它们的元素。字典属于映射类型。映射类型的元素由键（key）和值（value）组成，简称“键值对”。在一个映射型的变量中，键是唯一的。集合类型变量中的元素是没有顺序的，且不允许出现重复的元素，类似数学中的集合。

3.1　列表

列表（list）是 Python 中最常用的序列类型。它由一系列元素（又称数据项）组成，所有元素都被包含在一对方括号“[]”中。列表中的元素可以是任何类型的数据，并且不需要所有元素具有相同的类型。列表具有如下特性：

（1）列表中的元素类型可以相同，也可以不同。

（2）每个列表元素都有索引和值两个属性，索引用于标识元素在列表中的位置，值指的是元素本身的值。

序列类型的元素索引分为正索引和负索引两种情况（见图 3.1）。正索引的索引值从 0 开始，在列表中从左向右依次递增 1，因此最后一个元素的索引为元素个数 - 1。负索引的索引值从-1 开始，在列表中从右向左依次递减 1。一般情况下列表的索引是正索引。

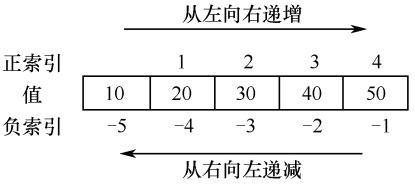


图 3.1　元素的正索引和负索引

3.1.1　创建列表和存取列表元素

可以使用方括号“[]”创建一个列表对象，也可以使用函数 list()创建一个列表对象。创建列表的代码格式是

```
[元素 1，元素 2，… ]
```

或

```
list(元素 1，元素 2，…)
```

【例】列表的创建。

```
In: lst = [ ]                    # 创建空列表
In: lst
Out: []

In: list()                       # 创建空列表
Out: []

In: [12, 3.0, 'python']          # 列表元素的数据类型不相同
Out: [12, 3.0, 'python']
```

```
In: list('python')                # 字符串中的每个字符成为列表中的一个元素
Out: ['p', 'y', 't', 'h', 'o', 'n']
```

3.1.2 列表基本操作

1. 访问列表元素

可以使用索引访问列表中的元素，其格式如下：

```
列表对象[index]
```

index：元素索引。

【例】访问列表中的元素。

（1）获取 lst 列表中索引为 1 的元素

```
In: lst = [1, 2, 3]
In: lst[1]
Out: 2
```

（2）修改 lst 列表中索引为 1 的元素的值

```
In: lst = [1, 2, 3]
In: lst[1] = 5                    # 将索引为 1 的元素的值修改为 5
In: lst
Out: [1, 5, 3]
```

2. 列表合并

可以使用运算符“+”将两个列表合并在一起。

【例】合并两个列表。

```
In: [1, 2] + [3, 'abc']
Out: [1, 2, 3, 'abc']
```

3. 重复

可以使用运算符“*”创建具有重复元素的列表。

【例】创建元素“1，2”重复 3 次的列表。

```
In: [1, 2] * 3
Out: [1, 2, 1, 2, 1, 2]
```

4. 迭代

迭代操作可用于遍历列表中的元素。

【例】迭代输出列表 lst 中的元素。

```
In: lst = [1, 2, 3, 4, 5]
In: for x in lst:
        print(x)
1
2
3
4
5
```

5. 成员判断

使用“in”操作符判断对象是否属于列表。

【例】判断对象是否属于列表。

```
In: 2 in [1, 2, 3]
Out: True
In: 5 in [1, 2, 3]
Out: False
```

3.1.3　列表常用函数

Python 提供了一系列处理列表对象的函数，可以对列表进行元素添加、删除、排序和反转等操作。表 3.1 列出了列表的常用函数。

表 3.1　列表的常用函数

函　数　名	说　　明
append(x)	在尾部添加新元素 x
index(x)	返回 x 在列表中的索引位置，不含 x 将返回一个运行时错误
count(x)	统计指定元素值 x 出现的次数，不含 x 将返回 0
del 对象[index]	删除列表对象中索引为 index 的元素
extend(lst)	添加新列表
insert(index, x)	将 x 插入到列表的 index 位置
len(lst)	求列表长度
pop()	删除元素，并返回删除对象
remove(x)	删除元素 x
reverse()	逆排列
sort()	排序

【例】列表常用函数的应用。

（1）在 lst 列表末尾添加元素

```
In: lst = [1, 2, 3]
In: lst.append(5)      # 末尾插入
In: lst
Out: [1, 2, 3, 5]
In: lst.index(1)       # 数据 1 在列表的第 0 个位置
Out:0
In: lst.index(4)       # 将报错，列表中不包含数据 4
ValueError: 4 is not in list
```

（2）统计 lst 列表中元素值 2 出现的次数

```
In: lst = [1, 2, 2, 2, 3]
In: lst.count(2)       # 2 出现了 3 次
Out: 3
In: lst.count(4)       # 列表中不含数据 4，返回 0 次
Out: 0
```

（3）删除 lst 列表索引为 1 的元素

```
In: lst = [1, 2, 3]
In: del lst[1]
In: lst
Out: [1, 3]
```

（4）将列表 x 中的元素添加到 lst 列表中

```
In: lst = [1, 2, 3]
In: x = [4, 5]
In: lst.extend(x)
In: lst
Out: [1, 2, 3, 4, 5]
```

（5）在 lst 列表索引为 1 的位置插入值为 5 的元素

```
In: lst = [1, 2, 3]
In: lst.insert(1, 5)
In: lst
Out: [1, 5, 2, 3]
```

（6）获取 lst 列表的长度（即列表元素的个数）

```
In: lst = [1, 2, 3]
In: len(lst)
Out: 3
```

（7）删除 lst 列表指定索引位置的元素，并返回该删除元素；若没有指定索引，则删除列表末尾的元素

```
In: lst = [1, 2, 3]
In: lst.pop(1)
Out: 2
In: lst
Out: [1, 3]
```

（8）删除 lst 列表中值为 2 的元素，如果有多个，那么只删除第一个

```
In: lst = [1, 2, 2, 3]
In: lst.remove(2)
In: lst
Out: [1, 2, 3]
```

（9）将 lst 列表中的元素反转

```
In: lst = [1, 2, 5, 4]
In: lst.reverse()
In: lst
Out: [4, 5, 2, 1]
```

（10）对 lst 列表中的元素进行排序，若是数值型数据则按从小到大排序，若是字符串则按字典顺序排序。sort()函数的参数 reverse=True 时为逆序排序，默认为 reverse=False 升序排序

```
In: lst = [1, 4, 3, 5, 2]
In: lst.sort()                          # 默认升序
In: lst
Out: [1, 2, 3, 4, 5]
In: lst = [1, 4, 3, 5, 2]
In: lst.sort(reverse=True)              # 参数 reverse=True 表示进行逆序排序
In: lst
Out: [5, 4, 3, 2, 1]
```

注意上式不能写为 lst=lst.sort()，因为 lst.sort()执行后未返回新列表（返回值为 None），若将 None 值赋给 lst，则 lst 原来的数据就会丢失。

lst.sort()方法直接改变原列表的顺序，如果排序时希望原列表不变，返回一个新的有序列表，那么可以借助 sorted()函数。

```
In: lst2 = sorted(lst)                # 执行后 lst 不变，得到新的有序列表 lst2
```

3.1.4　切片

列表的切片操作可以获得列表的多个元素。切片操作的基本格式如下：

```
列表对象[start : end : step]
```

start：索引开始位置，可以省略，默认为 0。

end：索引结束位置（不包括 end），可以省略，默认为列表长度。

step：步长，表示截取数据的步长，正数表示从左向右截取，负数表示从右向左截取，可以省略但不能为 0，默认为 1。

【例】列表切片应用。

```
In: lst = [1, 2, 3, 4, 5]
In: lst[1:4]            # 截取列表 lst 中从索引 1 到索引 3 的元素
Out: [2, 3, 4]
In: lst[1:]             # 截取列表 lst 中从索引 1 到末尾的元素
Out: [2, 3, 4, 5]
In: lst[:4]             # 截取列表 lst 中从开始到索引 3 的元素
Out: [1, 2, 3, 4]
In: lst[1:4:2]          # 截取列表 lst 中从索引 1 到索引 3，步长为 2 的元素
Out: [2, 4]
In: lst[-4:-1]          # 截取列表 lst 中从索引-4 到索引-2 的元素
Out: [2, 3, 4]
In: lst[3:0:-1]         # 步长为负，从右向左截取列表 lst 中从索引 3 到索引 1 的元素
Out: [4, 3, 2]
In: lst[-2:-5:-1]       # 步长为负，从右向左截取列表 lst 中从索引-2 到索引-4 的元素
Out: [4, 3, 2]
```

3.1.5　列表生成方式

1．产生一个数值递增列表

使用 range()函数可以产生一个数值递增且可迭代操作的对象，基本格式如下：

```
range(start, end, step)
```

start：起始元素值，可以省略，默认为 0。

end：结束元素值（不包括 end），不能省略。

step：步长，正数表示数值递增，负数表示数值递减，可以省略但不能为 0，默认为 1。

注意 range(100)并未在内存中立即产生 100 个数，只是产生了一个可迭代对象，具体数据将在后续迭代过程中逐一产生，这样有利于节省内存。将 range()函数产生的迭代对象作为列表创建函数 list()的参数，可以产生一个数值递增的列表。

【例】产生一个数值递增列表。

```
In: it = range(1, 5)        # 产生一个数值从 1 到 4 的迭代对象 it
In: lst = list(it)          # 以迭代对象为参数创建列表
In: lst
Out: [1, 2, 3, 4]
```

2．产生多维列表

多维列表可视为列表中嵌套的列表。例如，二维列表可视为每个元素都是一维列表的列表，三维列表可视为每个元素都是二维列表的列表。

【例】创建一个二维列表。

```
In: lst = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

此时，二维列表 lst 的内容如下所示，每行是一个一维列表：

1	2	3	4
5	6	7	8
9	10	11	12

多维列表的维数是从 0 开始的。例如，二维列表有第 0 维和第 1 维，分别对应于行和列。访问多维列表中的元素时，使用的命令格式如下：

```
列表对象[row][col]…
```

其中第一个“[]”为列表的第 0 维，第二个“[]”为第 1 维，以此类推。

【例】多维列表的访问。

```
In: lst = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
In: lst[1][2]                # 获取第 0 维索引为 1，第 1 维索引为 2 的数据
Out: 7
In: lst[1]                   # 获取第 0 维索引为 1 的数据，是一个一维列表
Out: [5, 6, 7, 8]
```

3．列表生成式

Python 的列表还可以用其特有的列表生成式语法产生。列表生成式的一般格式如下：

```
新列表对象=[ 表达式  for  变量  in  可迭代对象 <if 条件> ]
```

它表示从可迭代对象中取得变量的值，将变量代入表达式计算得到新值，由这些新值构成新的列表。<if 条件>是可选的，如果设有筛选条件，那么满足条件的变量值才能加入新列表。

```
In: lst = list(range(5))
In: lst
Out: [0, 1, 2, 3, 4]
In: lst2 = [x**2 for x in lst]
      # 列表生成式，将 lst 列表中的每个元素平方，得到列表 lst2
In: lst2
Out: [0, 1, 4, 9, 16]
In: lst3 = [ x for x in lst2 if x not in lst ]
      # 抽取在 lst2 中但不在 lst 中的元素得到列表 lst3
In: lst3
Out: [9, 16]
In: [ x for x in range(20) if x%3==0 ]  # [0, 20]内的 3 的倍数
Out:[0, 3, 6, 9, 12, 15, 18]
```

3.2 元组

元组（tuple）是用一对圆括号“()”定义的序列。元组中的元素值自元组创建后就不能修改。元组中的元素数据类型可以相同，也可以不相同。

3.2.1 创建元组和存取元组元素

1．创建元组

可以使用圆括号“()”定义创建一个元组对象，也可以使用函数 tuple()创建一个元组对象。

元组创建的格式如下：

```
    (元素 1，元素 2，…)          # ()可省略
```

或

```
    tuple(可迭代对象)            # 可迭代对象：字符串、列表、元组、字典、集合等
```

【例】创建三个元素的元组。

```
In: tpl = (1, 2, 3)              # 使用“()”创建元组
In: tpl
Out: (1, 2, 3)

In: tpl2 = tuple([1, 2, 3])      # 使用 tuple()函数创建元组，参数为列表对象
In: tpl2
Out: (1, 2, 3)

In: it = range(1, 5)             # 产生可迭代对象
In: tpl = tuple(it)              # 使用 tuple()函数创建元组，参数为可迭代对象
In: tpl
Out: (1, 2, 3, 4)
```

2．索引和切片

元组的索引和切片操作与列表的索引和切片操作非常相似。

【例】元组的索引操作和切片操作。

（1）访问元组 tpl 中索引为 1 的元素

```
In: tpl = (1, 2, 3)
In: tpl[1]
Out: 2
```

（2）截取元组 tpl 中从索引位置 1 到索引位置 3 的元素

```
In: tpl = (1, 2, 3, 4, 5)
In: tpl[1:4]
Out: (2, 3, 4)
```

3．其他操作

在 Python 中，元组的其他操作，如求元组的长度、合并、重复、迭代、成员判断等，也与列表的对应操作类似。

【例】元组的操作。

（1）元组求长度

```
In: tpl = (1, 2, 3)
In: len(tpl)
Out: 3
```

（2）合并两个元组

```
In: (1, 2, 3) + (4, 5)
Out: (1, 2, 3, 4, 5)
```

需要注意的是，由于元组中的元素是不能增删的，因此合并元组不是在已有的元组中添加元素，而是产生一个新元组，新元组与已有元组在内存中是独立的。

（3）重复，创建一个元素“1，2”重复 3 次的元组

```
In: (1, 2) * 3
Out: (1, 2, 1, 2, 1, 2)
```

（4）迭代元组中的所有元素

```
In: tpl = (1, 2, 3)
In: for x in tpl:
      print(x)
1
2
3
```

（5）判断对象是否为元组中的元素

```
In: 2 in (1, 2, 3)
Out: True
In: 5 in (1, 2, 3)
Out: False
```

（6）统计 tpl 元组中元素值 2 出现的次数

```
In: tpl = (1, 2, 2, 2, 3)
In: tpl.count(2)
Out: 3
```

3.2.2 元组和列表的差异

元组一旦创建，其中的元素就不可修改。元组也不能增加或删除元素。因为元组结构简单，所以元组占用的内存比列表占用的内存少，如果待处理的数据多用于查询，无须修改，那么可以考虑用元组存储。另外，由于元组是不可修改的，在函数调用或返回函数值时可考虑传递元组，以避免参数被误修改。

```
In: tpl = (1, 2, 3)
In: tpl[0] = 100            # 将报错，元组中的元素不可修改
TypeError: 'tuple' object does not support item assignment
In: tpl.append(4)           # 将报错，元组不支持 append()方法
AttributeError: 'tuple' object has no attribute 'append'
```

虽然不可以直接修改元组中的元素，但如果需要改变元组中的元素，那么可以先将元组转换为列表，然后对列表进行修改，再将修改好的列表转换为新的元组。列表和元组相互转换的函数是 tuple(lst)和 list(tpl)。

【例】将 tpl 元组中索引位置 1 的值由 2 改为 5。

```
In: tpl = (1, 2, 3)
In: lst = list(tpl)             # 将元组转换为列表
In: lst[1] = 5
In: tpl = tuple(lst)            # 将列表转换为元组
In: tpl
Out: (1, 5, 3)
```

3.2.3 序列操作函数

针对列表和元组等序列类型，Python 提供了一些实用的序列操作函数，如表 3.2 所示。

表 3.2　常用的序列操作函数

函　数　名	功　　能
all(seq)	判断序列对象的每个元素是否都为逻辑真值，返回 True 或 False
any(seq)	判断序列对象是否有逻辑真值的元素，返回 True 或 False
range(start,stop[, step])	返回从 start（默认值为 0）开始，到 stop 结束（不包括 stop）的可迭代对象，step 为步长（默认值为 1）
reversed()	反转序列，返回可迭代对象
sorted(iter)	对序列对象 iter 进行排序，返回一个有序列表
zip(iter1[, iter2,…])	将多个迭代对象相同位置的元素聚合成元组，返回一个以元组为元素的可迭代对象

【例】常用的序列操作函数。

```
In: all([1, 2, 'Python'])        # 非 0 元素对应布尔值 True，本例所有元素都对应 True
Out: True
In: all([0, 1, 2, 'Python'])     # 0 对应的布尔值为 False
Out: False
In: any([0, 1, 2, 'Python'])
Out: True
In: lst = [3, 2, 5, 4]
In: sorted(lst)                  # 排序
Out: [2, 3, 4, 5]
In: r = reversed(lst)            # 反转，生成可迭代对象
In: list(r)                      # 由可迭代对象创建列表
Out: [4, 5, 2, 3]
In: z = zip(['a', 'b', 'c'], [1, 2, 3])  # 产生一个以两个列表元素聚合成元组的新迭代对象
In: list(z)                       # 由可迭代对象创建列表
Out: [('a', 1), ('b', 2), ('c', 3)]
```

3.3　字典

字典（dict）可视为键值对构成的数据容器。搜索字典中的元素时，首先查找键，然后根据找到的键获取对应的值。这是一种高效、快速的查找方法。字典中的键必须是唯一的。

3.3.1　创建字典和存取键值对

1．创建字典

可以使用标记“{}”创建一个字典对象，也可以使用函数 dict()创建一个字典对象。字典中的每个元素都包含键和值两部分内容，创建字典时，键和值用冒号“:”隔开，字典的元素之间用逗号“,”分开。创建字典的格式如下：

```
{键 1:值 1, 键 2:值 2, …}
```

或

```
dict(键 1=值 1, 键 2=值 2, …)
```

【例】创建字典。

```
In: di = {}        # 空字典
In: di
```

```
Out: {}
In: dict()          # 空字典
Out: {}
In: di = {'学号': '001', '姓名': 'zhangsan', '年龄': 19}
In: di
Out: {'学号': '001', '姓名': 'zhangsan', '年龄': 19}
```

2. 字典的访问

字典中元素的访问是通过键获取相应的值。访问字典元素的格式如下：

```
字典对象[key]          # key：字典中元素的键
```

【例】访问字典中的元素。

```
In: dct = {'学号': '001', '姓名': 'zhangsan', '年龄': 19}
In: dct['姓名']        # 通过键取值
Out: 'zhangsan'
In: dct['001']         # 将报错，不能直接取出值。字典中没有名为'001'的键
KeyError: '001'
```

3. 添加和修改字典元素

添加和修改字典元素的格式如下：

```
字典对象[key] = value
```

key：元素的键。

value：元素的值。

如果字典中不存在 key 键，那么在字典中添加一个 key：value 的元素，如果字典中存在 key 键，那么将字典中 key 键对应的值修改为 value 值。

【例】添加、修改字典元素。

```
In: dct = {'学号': '001', '姓名': 'zhangsan', '年龄': 19}
In: dct['性别'] = '男'          # 在字典中添加键为“性别”，值为“男”的数据
In: dct
Out: {'学号': '001', '姓名': 'zhangsan', '年龄': 19, '性别': '男'}

In: dct['年龄'] = 20            # 修改字典中“年龄”的值为 20
In: dct
Out: {'学号': '001', '姓名': 'zhangsan', '年龄': 20, '性别': '男'}
```

3.3.2 字典的常用方法

Python 提供了许多专门处理字典对象的函数，表 3.3 列出了其中比较常用的函数。

表 3.3 字典常用函数

函 数 名	说 明
clear()	清除字典中的所有元素
copy()	复制整个字典
del 对象[key]	删除字典中键为 key 的元素
get(key, default)	返回键 key 对应的值，如果字典中不存在 key 键，那么返回 default
items()	返回所有键值对

（续表）

函 数 名	说　明
keys()	返回所有键
values()	返回所有值
update(dct)	使用 dct 字典数据更新当前字典
pop(key, default)	返回键 key 对应的值，并删除该元素，若字典中不存在 key 键，则返回 default

【例】字典常用函数的应用。

（1）清除字典对象中的所有元素

```
In: dct = {'学号': '001', '姓名': 'zhangsan', '年龄': 19}
In: dct.clear()
In: dct
Out: {}
```

（2）复制字典对象 dct

```
In: dct = {'学号': '001', '姓名': 'zhangsan', '年龄': 19}
In: d = dct.copy()
In: d
Out: {'学号': '001', '姓名': 'zhangsan', '年龄': 19}
```

（3）删除字典对象中的元素

```
In: dct = {'学号': '001', '姓名': 'zhangsan', '年龄': 19}
In: del dct['年龄']
In: dct
Out: {'学号': '001', '姓名': 'zhangsan'}
```

（4）获得字典对象中键对应的值

```
In: dct = {'学号': '001', '姓名': 'zhangsan', '年龄': 19}
In: dct.get('年龄')                    # 获取字典中键为“年龄”的值
Out: 19
In: dct.get('性别', '男')              # 字典中无“性别”键，默认返回“男”
Out: '男'
```

（5）获得字典对象中键对应的值，并删除该元素

```
In: dct = {'学号': '001', '姓名': 'zhangsan', '年龄': 19}
In: dct.pop('年龄')                    # 返回字典中键为“年龄”的值，并删除该元素
Out: 19
In: dct
Out: {'学号': '001', '姓名': 'zhangsan'}
```

（6）获得字典对象中的所有键

```
In: dct = {'学号': '001', '姓名': 'zhangsan', '年龄': 19}
In: dct.keys()                         # 获取字典中的所有键
Out: dict_keys(['学号', '姓名', '年龄'])
```

（7）获得字典对象中的所有值

```
In: dct = {'学号': '001', '姓名': 'zhangsan', '年龄': 19}
In: dct.values()                       # 获取字典中的所有值
dict_values(['001', 'zhangsan', 19])
```

（8）获得字典对象中的所有键值对

```
In: dct = {'学号': '001', '姓名': 'zhangsan', '年龄': 19}
In: dct.items()                        # 获取字典中的所有键值对
```

```
Out: dict_items([('学号', '001'), ('姓名', 'zhangsan'), ('年龄', 19)])
```

（9）更新字典 dct 中的元素

```
In: dct = {'学号': '001', '姓名': 'zhangsan', '年龄': 19}
In: d = {'学号': '002', '年龄': 20}
In: dct.update(d)                    # 根据字典 d 中的数据修改字典 dct
In: dct
Out: {'学号': '002', '姓名': 'zhangsan', '年龄': 20}
```

3.4 集合

集合（set）是由 0 个或多个元素构成的无序组合，集合中的元素不允许重复。集合是可变的，即可以添加或删除集合中的元素。集合不支持索引操作。

3.4.1 创建集合

可以使用标记“{}”创建一个集合对象，也可以使用函数 set()创建一个集合对象。集合中的元素用逗号“,”分隔。与字典中的每个元素都包含键和值两部分内容不同，集合中的每个元素只有值，没有键。创建集合的格式如下：

```
{元素 1，元素 2，…}
```

或

```
set(可迭代对象)
```

【例】创建集合。

```
In: set()                          # 创建空集合
Out: set()

In: {'C++', 'Java', 'Python'}    # 创建 3 个元素的集合
Out: {'Java', 'C++', 'Python'}   # 集合元素是无序的，此处的显示顺序和创建顺序不一致

In: set(['C++', 'Java', 'Python'])   # 以列表为参数，创建 3 个元素的集合
Out: {'Java', 'C++', 'Python'}
In: set([1, 1, 3, 3, 5, 5, 5])       # 集合不允许重复元素，所以可以利用集合快速去重
Out:{1, 3, 5}
```

创建空集合只能使用 set()函数而不能使用“{}”方式，因为“{}”创建的是空字典而不是空集合。

3.4.2 遍历集合

由于集合中的元素没有索引的属性，同时也没有像字典中“键值对”的对应关系，因此无法直接获取集合中的指定元素，只能通过迭代来遍历集合中的所有元素。

【例】遍历集合 s。

```
In: s = {'C++','Java','Python'}
In: for x in s:
      print(x)          # 输出的顺序可能和创建时的顺序不一致
Python
Java
C++
```

3.4.3　集合操作函数

Python 提供了一系列对集合对象进行各种操作的函数。表 3.4 列出了常见的集合操作函数。

表 3.4　常见的集合操作函数

函　数　名	说　　明
add(x)	添加元素，若集合中不存在元素 x，则添加
clear ()	清除所有元素
copy()	复制整个集合
len(s)	返回集合 s 的元素个数
pop()	随机返回集合中的一个元素，并删除该元素
remove(x)	删除元素 x，若集合中不存在 x，则报错
discard(x)	删除元素 x，即使 x 不存在也不报错
update(s)	将另一个集合的元素添加进来

【例】集合操作的常用函数的应用。

（1）向集合中添加元素

```
In: s = {'C++', 'Java', 'Python'}
In: s.add('PHP')
In: s
Out: {'PHP', 'Java', 'C++', 'Python'}
```

（2）删除集合中的元素

```
In: s = {'C++', 'Java', 'Python'}
In: s.remove('Java')              # 使用 remove()函数删除元素
In: s
Out: {'C++', 'Python'}
In: s.remove('Java')              # 再次删除，集合中已无 Java 元素将报错
KeyError: 'Java'
In: s.discard('Java')             # 使用 discard()删除，无 Java 元素也不报错
```

（3）清除集合中的所有元素

```
In: s = {'C++', 'Java', 'Python'}
In: s.clear()                     # 使用 clear()函数清除元素后的集合为空集合
In: s
Out:set()
```

（4）复制整个集合

```
In: s = {'C++', 'Java', 'Python'}
In: x = s.copy()
In: x
Out: {'Java', 'C++', 'Python'}
```

（5）将集合 t 中的元素添加进来

```
In: s = {'C++', 'Java', 'Python'}
In: t = {'PHP', 'C++'}
In: s.update(t)                          # 根据集合 t 的数据更新集合 s 的元素
In: s
Out: {'PHP', 'C++', 'Java', 'Python'}
```

（6）随机返回集合中的一个元素

```
In: s = {'C++', 'Java', 'Python'}
In: s.pop()                              # 随机返回一个元素，并在集合中删除该元素
Out: 'Java'
In: s
Out: {'C++', 'Python'}
```

（7）获取集合元素个数

```
In: s = {'C++', 'Java', 'Python'}
In: len(s)
Out: 3
```

3.4.4 集合运算：并、交、差

与数学中的集合概念一样，Python 中的集合也支持两个集合的并集、交集、差集等各种运算。常见的集合运算符如表 3.5 所示。

表 3.5 常见的集合运算符

运 算 符	说 明
S & T	交集，返回一个由两个集合 S 和 T 中都存在的元素构成的集合
S | T	并集，返回一个包含了两个集合 S 和 T 中的所有元素的集合
S - T	差集，返回一个由在集合 S 中存在但在集合 T 不存在的元素构成的集合
S ^ T	对称差集，返回一个由不在集合 S 和集合 T 中同时存在的元素构成的集合
S == T	集合 S 和集合 T 中的元素相同返回 True，否则返回 False
S != T	集合 S 和集合 T 中的元素不相同返回 True，否则返回 False
S <= T	子集测试，集合 S 是集合 T 的子集返回 True，否则返回 False
S < T	真子集测试，集合 S 是集合 T 的真子集返回 True，否则返回 False
S >= T	超集测试，集合 S 是集合 T 的超集返回 True，否则返回 False
S > T	真超集测试，集合 S 是集合 T 的真超集返回 True，否则返回 False

【例】常见的集合运算符的应用。

```
In: s = {10, 20, 30}
In: t = {20, 30, 40}
In: s & t                          # 交集
Out: {20, 30}
In: s | t                          # 并集
Out: {20, 40, 10, 30}
In: s - t                          # 差集
Out: {10}
In: s ^ t                          # 对称差集
Out: {40, 10}
In: s == t                         # 判断集合元素是否相同
Out: False
In: s != t                         # 判断集合元素不相同
Out: True
In: s = {10, 20, 30}
In: t = {10, 20}
```

```
In: s > t                          # 真超集测试
Out: True
In: s < t                          # 真子集测试
Out: False
```

3.5　可变类型和不可变类型

程序中的数据必须以特定类型存放在内存的某个区域中。根据变量对应的内存区域是否可以修改，Python 将数据类型分为不可变类型和可变类型。如果变量的值发生改变后对应的内存地址也发生改变，那么这种数据类型称为不可变数据类型。如果变量的值发生改变后对应的内存地址保持不变，那么这种数据类型称为可变数据类型。

Python 中的整型、浮点型、字符串型和元组属于不可变数据类型，列表、字典和集合属于可变数据类型。

【例】不可变数据类型的应用。

```
In: a = 10
In: id(a)                          # id()返回 a 的唯一标识符，等同于 a 的内存地址
Out: 262719776                     # 变量 a 的初始内存地址
In: a = 20                         # 改变 a 的值
In: id(a)
Out: 262719936                     # 变量 a 的内存地址发生改变
```

上面的代码中，语句“a = 20”并不是在变量 a 原来的内存区域（存放 10 的区域）上修改值，而是在另外一个内存区域存放“20”，因此变量 a 的内存地址发生了改变。注意，Python 的整型是不可变数据类型，并不是说整型变量赋值后就不可改变，而是说整型变量对应的内存区域不允许修改。为整型变量赋新值后，会使用新的内存空间存储新值，变量也指向新的内存区域。旧内存区域如果未被其他变量指向，那么会被废弃，并由 Python 自动回收内存空间。

【例】可变数据类型的应用。

```
In: lst = [1, 2, 3]
In: id(lst)
Out: 204715272                     # lst 的内存地址
In: lst.append(4)                  # lst 添加元素 4
In: lst[0] = 99                    # 修改 lst 中的第 0 个元素
In: lst
Out: [99, 2, 3, 4]
In: id(lst)
Out: 204715272                     # lst 的内存地址没有改变
```

上面的程序的执行结果表明，虽然使用 append()函数对列表 lst 添加了元素，也修改了第 0 个元素，但 lst 的内存地址并未改变，这表明是在原有内存中对旧值进行修改的。

【例】可变数据变量赋值时内存指向相同的问题。

```
In: lst = [1, 2, 3]
In: id(lst)
Out: 204715272                     # lst 的内存地址
In: lst2 = lst                     # 可变数据变量 lst 赋值给 lst2
In: id(lst2), id(lst)
Out: (204715272, 204715272)        # lst2 的内存地址与 lst 一样
```

```
In: lst2.append(4)                    # lst2 添加元素 4
In: lst2[0] = 99                      # 将 lst2 的第 0 个元素修改为 99
In: lst2
Out: [99, 2, 3, 4]
In: lst
Out: [99, 2, 3, 4]                    # lst 的元素也改变了
```

上面的程序显示，列表 lst 和 lst2 共享相同的内存地址，因此对 lst2 进行改变时，lst 也发生了改变，而这种改变可能并不是程序员所希望的。在对列表、字典这类可变数据对象进行操作时，要特别注意这种修改产生的副作用。

3.6 浅复制和深复制

为了解决可变变量赋值时两个对象指向同一内存区域从而产生修改时相互影响的问题，Python 提供了浅复制和深复制两种解决方法。

```
In: lst = [1, 2, 3]
In: lst2 = lst.copy()       # 浅复制，lst2 采用新存储空间保存赋值得到的 lst 的内容
In: id(lst),  id(lst2)      # 对比 lst 和 lst2 的内存地址
Out: (1979528444104, 1979527296520)       # 可见两个列表对象的内存地址不同
In: lst2.append(4)
In: lst2[0] = 99
In: lst2
Out: [99, 2, 3, 4]          # lst2 已变动
In: lst
Out: [1, 2, 3]              # 可见 lst 未变动

In: import  copy            # 引入 copy 模块
In: lst3 = copy.copy(lst)   # 也可利用 copy 模块中的 copy()函数完成浅复制
In: id(lst3), id(lst)
Out: (1979528442952，1979528444104)  # 两个列表对象的内存区域不同
```

浅复制是在赋值时开辟新的存储空间来保存新对象。新旧对象由于在内存中是独立的，所以不会出现修改时相互影响的问题。浅复制初步解决了内存指向相同的问题，但如果被复制的对象内部还含有可变元素，那么浅复制时内部的可变元素还是会指向相同的地址，示例代码如下：

```
In: lst = [1, 5, ['a', 'b']]         # lst 的第 2 个元素是一个可变元素['a', 'b']
In: lst2 = lst.copy()                # 浅复制
In: id(lst),  id(lst2)               # lst 和 lst2 的内存地址不同
Out: (1979528687752, 1979527288008)
In: lst2[0] = 99                     # 修改 lst2 的第 0 元素
In: lst2[2][1] = 'c'                 # 修改 lst2 中索引为 2 的元素（可变数据类型）
In: lst2
Out: [99, 5, ['a', 'c']]             # lst2 修改后的数据
In: lst
Out: [1, 5, ['a', 'c']]              # lst 的可变数据元素被改变，不可变数据元素未变
In: id(lst[2]),  id(lst2[2])         # 可见 lst 和 lst2 的第 2 个元素（列表）内存地址相同
Out: (1979527288904, 1979527288904)
```

从上面的程序的执行结果可以看出，浅复制的两个变量 lst 和 lst2 虽然内存地址不同，但

是如果变量中有可变数据元素，那么这些可变元素的内存地址还是相同的，改变一个可变元素的值，另一个可变元素也会受到影响。为了将两个对象的内存地址彻底分开，可借助 copy 模块中的 deepcopy()函数进行深复制。

```
import copy                              # 导入 copy 模块
In: lst = [1, 5, ['a', 'b']]            # lst 的第 2 个元素是一个可变元素['a', 'b']
In: lst3 = copy.deepcopy(lst)            # 使用 copy 模块的 deepcopy()函数进行深复制
In: lst3[0] = 99                         # 修改 lst3 的第 0 个元素
In: lst3[2][1] = 'c'                     # 修改 lst3 中索引为 2 的元素，可变数据类型
In: lst3
Out: [99, 5, ['a', 'c']]                 # lst3 修改后的数据
In: lst
Out: [1, 5, ['a', 'b']]                  # lst 的数据不变
In: id(lst[2]),  id(lst3[2])             # 深复制时，lst 和 lst3 的第 2 个元素内存地址不同
Out: (1979528442056, 1979528427912)
```

从上面的代码的执行结果可以看出，深复制的两个变量 lst 和 lst3 是完全独立的，改变 lst3 中的可变元素，lst 不会随之发生改变。

3.7　本章小结

1. 序列类型数据包括字符串、列表和元组。
2. 列表是一种可变序列类型，使用“[]”或 list()函数创建列表对象。通过添加、修改和删除改变列表中元素的值，可以索引或切片访问。
3. 元组是一种不可变序列类型，使用“()”或 tuple()函数创建元组对象，元组中的元素不能增删，元素值不能修改，可以索引或切片访问。列表和元组相互转换的函数是 tuple()和 list()。
4. 字典是一种映射类型，由键和值组成，是可变数据类型，其元素没有固定的顺序，不能索引访问。使用“{}”创建字典对象，通过键访问值。
5. 集合是一种无序且不重复的元素集，使用“{}”创建集合对象。与字典不同，集合中的元素不是“键值对”形式的。
6. 在 Python 语言中，数据类型可分为可变数据类型和不可变数据类型。
7. 对可变数据类型，变量之间的赋值可借助浅复制或深复制实现内存指向的独立。

习题

一、单项选择题

1. 下列选项中，属于元组操作的函数是（　　）

   A. pop()　　B. sort()　　C. reverse()　　D. count()

2. 下列选项中，不属于字典操作的函数是（　　）

   A. clear()　　B. keys()　　C. update()　　D. sort()

3. 下列选项中，不能使用索引运算的是（　　）

   A. 列表　　B. 元组　　C. 集合　　D. 字符串

4. 下列字典定义正确的是（　　）

   A. a = ['a', 1, 'b', 2, 'c', 3]　　B. b = ['a':1, 'b':2, 'c':3]

C. c = {a:1, b:2, c:3}　　D. d = {'a':1, 'b':2, 'c':3}

5. 代码 type({'a',1,'b',2,'c',3})的运算结果是（　　）

A. <class 'list'>　　B. <class 'tuple'>

C. <class 'dict'>　　D. <class 'set'>

6. 代码 len({1,2,2,2,3,4,5})的运算结果是（　　）

A. 1　　B. 3　　C. 5　　D. 8

7. 下列关于列表的说法中，错误的是（　　）

A. list 是一个有序集合，可以添加或删除元素

B. list 是不可变的数据类型

C. list 可以存放任意类型的元素

D. 使用 list 时，索引可以是负数

二、简答题

1. 简述列表的特性。
2. 简述元组的特性。
3. 简述字典的特性。
4. 简述集合的特性。
5. 列表和元组两种序列结构有何区别？
6. 列表、元组、字典和集合分别用什么符号或函数创建？
7. 集合运算符&、|、-、^、>=和<=分别表示什么运算？

三、程序题

1. 输出列表 lst=[1,2,1,12,10,5,2,7,1,8]中不重复的元素，并统计数据个数。
2. 假设两个元组 x=(1,3,2)和 y=(5,9,4,7)，将两个元组的数据合并再并按从小到大的顺序排序。
3. 假设两个集合 a={1,2,3,4,5}和 b={2,4,6}，找出属于集合 a 但不属于集合 b 的元素。
4. 使用字典保存学生姓名和对应成绩，输出所有学生姓名，并找出某个学生的成绩。

第 4 章　程序控制结构

计算机程序是一组描述某个问题的求解算法并能被计算机识别和执行的指令序列。程序控制结构是指计算机程序中以某种逻辑次序执行的一系列操作的语句组织结构。理论和实践证明，实现计算机求解问题的算法均可通过顺序、选择、循环三种基本控制结构构造出来，也就是说，任何程序中的语句块（由一条或多条语句组成的语句序列）结构都属于这三种结构之一。本章首先介绍 Python 语言的顺序、选择和循环三种控制结构的语法与使用方法，然后简单介绍 Python 异常处理结构的使用方法。

4.1　顺序结构

程序的顺序结构是指按语句出现的先后顺序执行的程序结构。程序设计语言并不提供专门的控制流语句来表达顺序控制结构，而用程序语句的自然排列（即从上到下、从左至右）顺序来表达。计算机按照此顺序逐条执行语句，即从顺序结构的第一条语句开始执行，每当一条语句执行完毕，计算机就自动转到下一条语句执行，直到完成顺序结构的最后一条语句的执行。

顺序结构的流程图的基本形式如图 4.1 所示，它表示计算机在执行语句块 1 后，接着执行语句块 2。

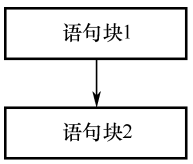


图 4.1　顺序结构的流程图

Python 程序的顺序结构不需要额外的程序流程控制语句，只需将程序中的多条语句按自然顺序编程即形成顺序结构。

如下程序段的功能是从键盘获取一个三位数，然后依次提取这个三位数的百位数、十位数和个位数，最后输出这三个数位的数字。程序段的语句将按从上到下的顺序依次执行。

```
shu = eval(input("输入一个三位数："))
bai = shu // 100
shi = (shu // 10) % 10
ge = shu % 10
print(bai, shi, ge)
```

【例 4.1】设计一程序从键盘读入圆的半径，完成圆周长和圆面积的计算并显示输出圆的周长与面积。

程序如下：

```
pi = 3.14159
r = eval(input('输入圆的半径：'))
d = 2 * pi * r
s = pi * r * r
print('d={:.2f}'.format(d))
print('s={:.2f}'.format(s))
```

4.2 选择结构

选择结构（也称分支结构）中包含条件判断语句（控制选择的条件表达式语句）和可选择执行的语句块。条件表达式可以是各种类型的表达式（如算术表达式、关系表达式、成员判别式等），当条件表达式的值为 False、0、空串、空列表、空元组、空字典、空集合时，条件表达式的逻辑值就为假；否则条件表达式的逻辑值便为真。计算机首先要计算条件表达式，然后根据条件表达式的逻辑值（真或假）来选择执行语句块。Python 的选择结构是通过 if 语句实现的，包括单分支、二分支和多分支三种基本形式。

4.2.1 二分支选择结构

二分支选择结构的流程图如图 4.2 所示。二分支选择结构的执行过程是，先计算条件语句中条件表达式 1 的值以判断其真假。若为真则执行语句块 1，若为假则执行语句块 2。

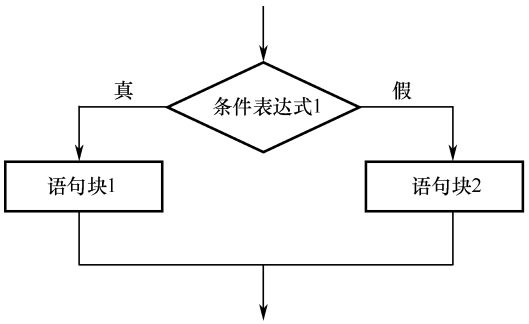


图 4.2 二分支选择结构的流程图

二分支选择结构的 if 语句的一般形式如下：

```
if 条件表达式 1:
    语句块 1
else:
    语句块 2
```

当条件表达式 1 的逻辑值为真时，计算机将执行语句块 1 而不再执行语句块 2。当条件表达式的逻辑值为假时，计算机将执行语句块 2 而不再执行语句块 1。根据条件表达式的不同形式，常见的二分支选择结构的 if 语句形式有如下 6 种。

条件表达式 1 是比较两个数形成的关系表达式。例如：

```
a, b = eval(input('输入两个数：'))
if  a >= b:                    # 按语法要求，此处必须有冒号
    print('a 大于等于 b')       # 下级语句块要缩进
else:                          # else 处也有冒号，不能缩进，要和 if 对齐
    print('a 小于 b')
```

条件表达式 1 是比较字符串形成的关系表达式。例如：

```
a = input('输入一个字母：')
if  'a' <= a <= 'z':
    print('输入的是小写字母')
else:
    print('输入的是大写字母')
```

条件表达式 1 是由逻辑运算符形成的逻辑表达式。例如：

```
x = eval(input('输入一个整数：'))
if  (x % 2 == 0) and (0 < x < 100):
    print('输入的是小于 100 的偶数')
else:
    print('输入的是其他数')
```

条件表达式 1 是一个函数调用，根据函数的返回值确定条件的真假。例如：

```
s = input('输入一个字符串：')
```

```
if  s.endswith('txt'):
    print('输入的字符串以 txt 结尾')
else:
    print('输入的字符串不以 txt 结尾')
```

下面的程序段的输出是“不是空列表”。因为列表 s 不是空列表，因此 if 语句的条件表达式的逻辑值为真。

```
s = [1, 2, 3]
if  s:
    print('不是空列表')
else:
    print('空列表')
```

而下面的程序段的输出是“空列表”。因为列表 s 在 if 语句前执行了清空操作使得 s 成了空列表，因此 if 语句的条件表达式的逻辑值为假。

```
s = [1,2,3]
s.clear()     # 清空列表
if  s:
    print('不是空列表')
else:
    print('空列表')
```

4.2.2　单分支选择结构

单分支选择结构的一般形式如下：

```
if 条件表达式 1:
    语句块 1
```

当条件表达式 1 的逻辑值为真时，计算机执行语句块 1，否则计算机将跳过语句块 1 而执行后续的语句。单分支选择结构的流程图如图 4.3 所示。

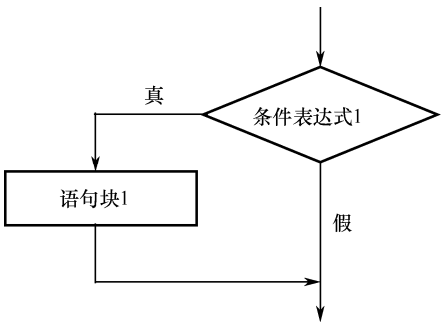


图 4.3　单分支选择结构的流程图

下面的程序段的功能是将输入的两个数按从大到小的顺序显示输出。

```
a, b = eval(input('输入两个数(逗号分隔): '))
if a < b:
    a, b = b, a
print(a, b)
```

按照从小到大的顺序输入两个数据时，if 语句中条件表达式的逻辑值为真，所以选择执行“语句块 1”（a, b = b, a）以交换两个变量的值。按照从大到小的顺序输入两个数据时，if 语句中条件表达式的逻辑值为假，此时不执行“语句块 1”（a, b = b, a）而跳至选择结构的后续语句 print(a, b)执行。

在前面的例子中，if 语句中的语句块 1 和语句块 2 都只有一个语句。实际上，Python 的一个语句块可以包含多个语句，并且多个语句的缩进应相同（即左对齐）。

【例 4.2】输入一个时间（小时:分钟:秒），输出该时间经过 15 分 37 秒后的时间。

程序如下：

```
hour, minute, second = input('input time(h:m:s):').split(':')
hour = int(hour)
minute = int(minute)
```

```
second = int(second)
second += 37
if  second >= 60:
    second = second-60 # 同一语句块中的语句缩进要一致
    minute += 1
minute += 15
if  minute >= 60:
    minute = minute-60
    hour += 1
if  hour == 24:
    hour = 0
print('%d:%d:%d' % (hour, minute, second))
```

程序中前面两个 if 语句的语句块都有两个语句。相邻的两个或多个语句具有相同的缩进时，就形成了一个语句块。

4.2.3 多分支选择结构

多分支选择结构的流程图如图 4.4 所示。多分支选择结构的执行流程规则如下：先判断表达式 1 的逻辑值是否为真，若为真则执行语句块 1，执行完语句块 1 后，跳过这个多分支结构中的其他所有语句而开始执行选择结构的后续语句；若为假则接着判断表达式 2 的逻辑值是否为真，若为真则执行语句块 2，执行完语句块 2 后，跳过这个多分支结构中的其他所有语句而开始执行选择结构的后续语句；若为假则接着判断表达式 3 的逻辑值是否为真……若所有条件表达式的逻辑值都为假，则选择执行 else 语句块 n+1。

Python 的多分支选择结构一般形式如下：

```
if 表达式 1:
     语句块 1
elif 表达式 2:
     语句块 2
…
elif 表达式 n:
     语句块 n
else:
     语句块 n+1
```

下面的代码是一个有三个分支的多分支选择结构程序段，它根据 a 和 b 的值输出不同的信息。

```
a, b = eval(input("输入两个整数："))
if a > b:
    print("a 大于 b")
elif a < b:
    print("a 小于 b")
elif a == b:
    print("a 等于 b")
```

【例 4.3】编写一个程序从键盘读入一个小于 100000 且大于 10 的整数 m ，若 m 是 k（$k \geq 2$）位的整数，则求出 m 的后 $k-1$ 位的数并输出该数。例如，若 m 的值为 6843，则程序输出 843；若 m 的值为 752，则程序输出 52。

程序如下：

```
m = eval(input("Input an integer:"))
```

```
if  m >= 10000:
    m = m % 10000
elif m >= 1000:
    m = m % 1000
elif m >= 100:
    m = m % 100
else:
    m = m % 10
print('m =', m)
```

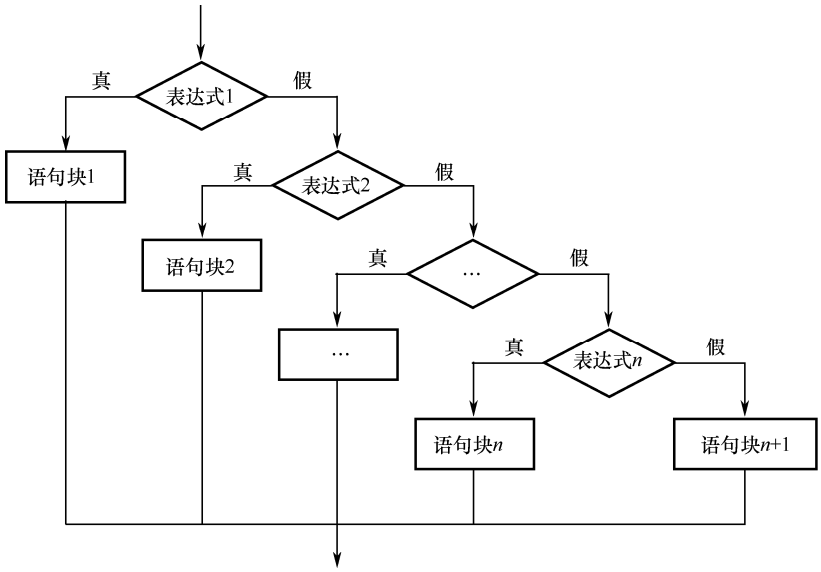


图 4.4　多分支选择结构的流程图

4.2.4　嵌套的选择结构

如果选择结构的语句块中包含选择结构，那么就形成了嵌套的选择结构。嵌套的选择结构的一般形式如下：

```
if 条件表达式 1:
    if 条件表达式 2:
        语句块 1
    else:
        语句块 2
    语句块 3
else:
    if 条件表达式 3:
        语句块 4
    else:
        语句块 5
    语句块 6
```

嵌套的选择结构可能会形成多层次的缩进，因此要注意确定不同层次语句块的缩进深度，否则会导致程序不能正确执行。

【例 4.4】编写程序从键盘读入一个大于 0 且小于 100000 的正整数，要求：①分别打印出它

的每一位数字；②输出它是几位数；③按逆序打印出各位数字。例如，若读入的是正整数 12345，则其逆序为 54321。

程序如下：

```
m = eval(input("Input an integer:"))
ge = m % 10
m = m // 10
wei = 1
ni = ge
print("个位数是", ge)
if m > 0:
    shi =  m % 10
    m = m // 10
    wei = 2
    ni = ni * 10 + shi
    print("十位数是", shi)
    if m > 0:
        bai = m % 10
        m = m // 10
        wei = 3
        ni = ni * 10 + bai
        print("百位数是", bai)
        if m > 0:
            qian = m % 10
            m = m // 10
            wei = 4
            ni = ni * 10 + qian
            print("千位数是", qian)
            if m > 0:
                wan = m
                wei = 5
                ni = ni * 10 + wan
                print("万位数是", wan)
print("输入数的位数是", wei)
print("它的逆序数是", ni)
```

4.3 循环结构

循环结构是程序中根据判定条件对一组语句可以重复执行多次的语句结构。采用循环结构是为了实现有规律的重复计算处理。循环结构有三个要素：循环变量、循环体和循环终止条件。循环结构的程序流程图如图 4.5 所示，图中条件判断框的两个出口分别对应条件成立和条件不成立时执行的语句，其中一个要指向循环体（语句块 1），然后从循环体回到判断框的入口处。因此循环结构可视为一个条件判断语句和一个向回转向语句的组合。循环结构的执行过程如下：先判断条件表达式 1，若其逻辑值为真则执行语句块 1，然后继续判断条件表达式 1……若条件表达式的逻辑值为假则退出循环而转去执行循环结构的后续语句。

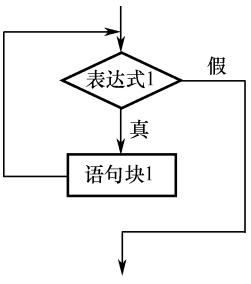


图 4.5 循环结构的流程图

Python 程序的循环结构包含 while 循环结构和 for 循环结构两种形式。

4.3.1　while 循环

while 循环的一般形式如下：

```
while 条件表达式:
    语句块 1
[else:
    语句块 2]
```

在 while 循环结构中，方括号中的 else 和语句块 2 是可选的，称为 while 循环的 else 子句。while 循环的执行规则如下：先判断条件表达式的逻辑值是否为真，若为真则执行语句块 1，然后重复判断表达式的逻辑值是否为真——执行语句块 1 的过程，直到条件表达式的逻辑值为假（也称循环条件不成立）时不再执行语句块 1。while 循环带有 else 子句时，当条件表达式为假而不再执行语句块 1 时，程序会接着执行 else 子句中的语句块 2。但是，如果在语句块 1 中出现了异常或在语句块 1 中执行 break 语句而退出了循环，那么不执行 else 子句中的语句块 2 而跳转执行循环结构的后续语句。

while 循环的循环条件可以有多种不同的形式。例如，下面的程序段的循环条件是比较整型变量的值：

```
i = 1
sum = 0
while  i <= 10:
    sum = sum + i
    i = i + 2
print("sum =", sum)
```

在以上程序段中，循环条件是变量 i 的值小于等于 10，循环体的功能是把变量 i 加到变量 sum 中。循环结束时，变量 i 的值为 11，sum 的值为 25。循环体中的语句 i = i + 2 的功能是改变变量 i 的值，使循环趋于结束。这样的语句对 while 循环结构非常重要，如果没有该语句，那么循环结构将永远不会结束而变成无限循环。

对于字符串和组合数据类型如列表、元组、集合、字典等，可以通过 while 循环和下标对其进行遍历。下面的程序段的循环条件是判断字符串的下标：

```
st = input('Input a String:')
i = 0
while  i < len(st):
    if  'a' <= st[i] <= 'z':
        print(st[i].upper(), end = '')        # 小写转换为大写
    else:
        print(st[i], end = '')
    i = i + 1
```

以上 while 循环的循环条件是判断变量 i 是否在字符串 st 的下标范围内，循环体的功能是判断字符是否为小写，若是则把小写字母转换为大写字母然后输出，否则直接输出。

下面的程序段的循环条件是判断列表的下标：

```
a = [1, 2, 3, 4, 5, 6]
i = 0
while  i < len(a):
```

```
        a[i] = a[i] + 10
        i = i + 2
    print(a)
```

以上 while 循环的循环条件是判断变量 i 是否在列表 a 的下标范围内，循环体的功能是把下标为偶数的列表元素值加上 10。执行后，输出为[11, 2, 13, 4, 15, 6]。

【例 4.5】编写程序，它的功能是逐个输入一系列字符，若输入的是英文字母则直接输出，若输入的是非英文字母则输出*字符，若输入的是回车则结束程序。

程序如下：

```
s = input('输入单个字符:')
while  len(s) > 0:
    if  s.isalpha():
        print(s)
    else:
        print('*')
    s = input('继续输入:')
```

【例 4.6】编写程序，输入一个整数 m（$m>10$），m 的位数是 n，求 m 的后 $n-1$ 位数，并输出该数。

程序如下：

```
m = eval(input('input a number:'))
s = 0
s1 = 1
t = m
while t > 10:
    p = t % 10
    s = s + p * s1
    s1 = s1 * 10
    t = t // 10
print('s =', s)
```

4.3.2 for 循环

for 循环适用于需要对组合数据或迭代对象的元素进行遍历的情况。for 循环还需要另外一个关键字 in 配合使用。for 循环的一般形式如下：

```
for 元素 in 组合数据或迭代对象:
    语句块 1
[else:
    语句块 2]
```

如果需要遍历某个确定范围内的数字，那么可以使用 range()函数。下面的程序段的功能是求 1 到 10 的和。

```
sum = 0
for i in range(1, 11):
    sum = sum + i
print('sum =', sum)
```

与 while 循环相比，for 循环的循环体内不需要用一个单独的语句来更改循环变量，因此显得更加简洁。

下面的程序段的功能是首先从键盘读入一个字符串，然后遍历该字符串中的每个字符。若字符是大写字母则转换为小写字母并输出，若是其他字符则直接输出。

```
st = input('Input a String:')
for  s  in  st:
    if  'A' <= s <= 'Z':
        print(s.lower(), end = '')
    else:
        print(s, end = '')
```

下面的程序段的功能是首先随机生成一个列表，列表的长度为 10，元素为 100 以内的随机正整数，然后遍历列表的元素并输出其中的奇数。

```
from random import randint
a = [randint(1,100) for i in range(10)]   # 用列表生成式产生随机数
for v in a:
    if v % 2 == 1:
        print(v, end=' ')
```

以上两个例子是对字符串或列表的元素进行遍历，在遍历过程中不需要通过下标就可以访问其中的每个元素，但在遍历的过程中不能得到元素的下标。如果在遍历过程中需要利用元素的下标信息，那么就要用到 enumerate()函数，该函数将返回一个由下标和值组成的迭代对象。

下面的程序段的功能是首先随机生成一个元组，元组的长度为 20，元素为 100 以内的随机正整数，然后遍历元组并输出其中的偶数及对应的下标。

```
from random import randint
a = tuple((randint(1,100) for i in range(20)))
for i, v in enumerate(a):
    if v % 2 == 0:
        print(i, '-', v)
```

【例 4.7】编写程序按照以下公式计算 s，并输出计算结果，其中 n 由用户输入。

$$s=1+1/(1+2)+1/(1+2+3)+\cdots+1/(1+2+3+\cdots+n)$$

程序如下：

```
n = eval(input('Input a number:'))
s = 1
t = 1
for i in range(2, n+1):
    t = t + i
    s = s + 1 / t
print('s =', s)
```

【例 4.8】一个班有 50 个学生，用一个列表保存所有学生数学课的成绩（成绩随机生成），分别统计不及格和及格学生的人数与平均分。

程序如下：

```
from random import randint
score = [randint(1,100) for i in range(50)]    # 用列表生成式产生随机成绩
ji = 0
jisum = 0
bu = 0
busum = 0
for s in score:
```

```
            if s >= 60:
                ji =  ji + 1
                jisum = jisum + s
            else:
                bu = bu + 1
                busum = busum + s
        jsavg = jisum / ji
        buavg = busum / bu
        print("及格人数为{}，平均分为{:.0f}".format(ji, jsavg))
        print("不及格人数为{}，平均分为{:.0f}".format(bu, buavg))
```

4.3.3 break 语句和 continue 语句

有时要在循环体的执行过程中终止当前循环过程，这时就要用到 break 语句和 continue 语句。break 语句的功能是结束循环，继续执行循环结构后续的语句。

下面的程序段的功能是从一个班的成绩表（保存在列表 score）中查找第一个优秀学生，并输出他的学号和成绩。找到一个学生的成绩满足优秀条件后，无须再查找其他学生的成绩，所以用 break 语句跳出循环。

```
        from random import randint
        score = [randint(1,100) for i in range(50)]
        for i, s in enumerate(score):
            if s >= 90:
                break
        print("第一个优秀学生是{}，他的成绩为{}".format(i,s))
```

continue 语句的功能是结束本次循环，跳过循环体中还没有执行的剩余语句，接着执行循环条件的判断以确定是否开启下一轮循环。

下面的程序段中，循环体的功能是输入一个数，若该数是负数则执行 continue 语句而跳过它后面的语句，即不输出该负数。若输入的是非负数则不会执行 continue 语句而执行输出语句，即输出该非负数。

```
        t = 1
        while  t <= 10:
            a = eval(input('Input a number:'))
            if  a < 0:
                continue
            print(a)
            t = t +1
```

break 语句和 continue 语句可以在同一个循环体中出现，例如：

```
        q = 1
        for p in range(1,100):
            if q >= 30:
                break
            if q % 3 == 1:
                q = q +3
                continue
            q = q - 5
        print("p = ", p)
```

该程序段的输出为 p = 11。

【例 4.9】斐波那契数列 $f(n)$ 的定义为：$f(1)=1, f(2)=1, f(n)=f(n-1)+f(n-2)$。编写程序，输入正整数 n，求斐波那契数列中大于 n 的最小一个数，并输出该数。

程序如下：

```
n = eval(input("Input a number: "))
a = 1
b = 1
c = 1
while True:
    if  c <= n:
        c = a+b
        a = b
        b = c
    else:
        break
print("c = ", c)
```

4.3.4　else 子句

当 while 循环和 for 循环正常终止时，即由于循环条件不成立而结束循环时，程序将执行循环的 else 子句。若由于执行 break 而结束，或在循环体中出现异常而结束，则不执行 else 子句。

例如，以下程序的功能是随机产生 10 个 0～20（包括 0）之间的整数，计算它们的倒数之和，并输出得到的和。

```
from random import randint
s = 0
i = 0
while i < 10:
    s = s + 1 / randint(0,20)
    i = i + 1
else:
    print('s=', s)
```

在上面的程序中，若产生的随机数没有 0，则能正常执行完全部程序，并输出得到的和。若产生的一个随机数是 0，则计算它的倒数时会抛出异常，而不会执行 else 子句中的输出语句。

4.3.5　循环的嵌套

若一个循环语句的循环体内又包含另一个完整的循环语句，则称为循环的嵌套，又称二层循环。内嵌的循环还可以嵌套另外的循环，即可以形成多层循环。

两语句（while 语句和 for 语句）构成的循环可以互相嵌套，具体分为以下几种情况。

（1）外层是 while 循环，内层可以是 while 或 for 循环，如图 4.6 所示。

```
while:
    #外层循环体
    while:
        #内层循环体
    …
```

```
while:
    #外层循环体
    for in:
        #内层循环体
    …
```

图 4.6　外层是 while 循环的嵌套

（2）外层是 for 循环，内层可以是 while 循环或 for 循环，如图 4.7 所示。

图 4.7　外层是 for 循环的嵌套

循环的嵌套中，内层循环的执行次数可能会受到外层循环的控制。

在下面的程序段中，内层循环的执行次数是受控于外层循环的。

```
for i in range(10):
    for j in range(i):
        print('*', end = ' ')
    print()
```

【例 4.10】随机产生一个列表，列表中包含 100 个 10000 以内的正整数，输出其中的素数及对应的下标。

程序如下：

```
from random import randint
a = [randint(2,10000) for i in range(100)]
for i, v in enumerate(a):
    j = 2
    while j < v:
        if v % j == 0:
            break
        j = j + 1
    else:
        print("第{}个数{}是素数".format(i, v))
```

【例 4.11】随机生成一个 10 行 10 列的二维数组，数组元素的值介于 1 和 9 之间，逐行输出该二维数组及每一行的和。

程序如下：

```
from random import randint
a = []
for i in range(10):
    b = [randint(1,9) for j in range(10)]
    a.append(b)
for p in a:
    s = sum(p)
    for q in p:
        print(q, end = ' ')
    print(s)
    print()
```

【例 4.12】随机生成一个 10 行 10 列的二维数组，数组元素的值介于 1 和 9 之间，逐行输出该二维数组及每一列的和。

程序如下：

```
from random import randint
a = []
for i in range(10):
    b = [randint(1,9) for j in range(10)]
    a.append(b)
sm = []
for j in range(len(a[0])):
    s = 0
    for i in range(len(a)):
        s = s + a[i][j]
    sm.append(s)
for p in a:
    for q in p:
        print(q, end = ' ')
    print()
for s in sm:
    print(s, end = ' ')
```

4.4　异常处理结构

异常处理（又称错误处理）通常是防止未知错误产生而采取的处理措施，它提供了处理程序运行时出现的任何意外或异常情况（即超出程序正常执行流程的某些特殊条件）的方法。Python 程序在执行过程中出现错误时会引发异常。例如，执行除法操作遇到除数为 0 时，Python 系统就会因无法处理该错误而导致程序运行崩溃。异常处理结构可以对程序中的各种未知错误进行处理，以防止程序崩溃。

Python 异常处理结构的一般形式如下：

```
try:
    语句块 1
except Exception:
    语句块 2
[else:
    语句块 3]
[finally:
    语句块 4]
```

其中 else 子句和 finally 子句是可选的。异常处理结构的执行流程如下：首先执行语句块 1；若在语句块 1 中出现了异常，则中断语句块 1 的执行而转去执行语句块 2；若在语句块 1 中未出现异常且有 else 子句，则执行语句块 3。若有 finally 子句，则 finally 子句中的语句块 4 无论语句块 1 中有没有错误都会执行。语句块 1 中没有错误时，执行完语句块 3 后就执行语句块 4；语句块 1 中有错误时，执行完语句块 2 后就执行语句块 4。

4.3.4 节中示例程序的功能是随机产生 10 个 0～20（包括 0）之间的整数，计算它们的倒数之和，并输出得到的和。当产生的随机数是 0 时，就会出现除 0 错误而导致程序崩溃。为了防止

程序崩溃，可以采用异常处理结构来处理该错误。例如：

```
from random import randint
sm = 0
i = 0
while i < 10:
    try:
        sm = sm + 1 / randint(0,20)
        i = i + 1
    except Exception as e:
        print("错误:",  e)
else:
    print(sm)
```

出现除 0 错误时，except 子句会捕获该错误并输出错误提示，然后继续执行，直到程序正常结束。

在以下程序段中：

```
try:
  a = [1, 2, 3]
  i = 1
  print("a[{}] = {}".format(i, a[i]))
except  Exception:
    print("error")
else:
    print("correct")
finally:
    print("always execute")
```

当 try 的语句块中没有出现错误时，就执行 else 子句和 finally 子句中的语句块。所以执行以上代码时，输出如下：

```
a[1] = 2
correct
always execute
```

当 try 的语句块中出现错误时，就执行 except 子句和 finally 子句中的语句块。所以将上面的代码中的 i=1 改为 i=3 再执行时，由于下标越界程序将出错，输出如下：

```
error
always execute
```

4.5 本章小结

本章介绍了程序设计的三种基本结构，即顺序结构、选择结构和循环结构。在 Python 中将语句按顺序书写即可实现顺序结构。用 if 语句实现选择结构，用 while 和 for 语句实现循环结构。在循环语句中可以使用 continue 语句和 break 语句实现循环中的跳转功能。在 Python 中，循环语句可以有 else 子句。本章还介绍了实现多重循环的循环嵌套，重点介绍了二重循环的实现。最后本章介绍了 try 异常处理结构。

习题

1. 已知函数 $f(x)=5x^3+10x^2-27x-31$，输入自变量 x，输出函数值 $f(x)$。
2. 输入两个两位数的正整数 a 和 b，将它们合并成一个整数放在 c 中。合并的方式如下：将 a 数的十位和 b 数的十位依次放在 c 数的千位和百位上，将 a 数的个位和 b 数的个位数依次放在 c 数的十位和个位上。输出合并后的整数 c。
3. 编写程序输入三个实数值，计算函数值 $F(x,y,z)=(x-y)/(x+y)+(z-y)/(z+y)$，其中 x 和 y 的值不等，z 和 y 的值不等，输出函数值。
4. 输入一个表示星期的数字（1 表示星期一……6 表示星期六，7 表示星期日），输出对应的星期英文单词。
5. 编写程序输入三个整数，最小的放在 a 中，最大的放在 c 中，中间的放在 b 中。
6. 编写程序进行字母转换。若输入的字母为小写英文字母，则转换为相应的大写英文字母；若输入的字母是大写英文字母，则转换为相应的小写英文字母，分别输出转换前、后的字母。
7. 编写程序，根据输入的三个边长值，判断能否构成三角形。若能构成等边三角形则输出 1，若能构成等腰三角形则输出 2，若能构成一般三角形则输出 3，若不能构成三角形则输出 0。
8. 设计程序，根据以下的对应关系，对输入的每个 x 值，求 y 的值。

$$y=\begin{cases}(x+1)(x-2), & x>2\\ 3x+4, & -1<x\le 2\\ x+1, & x\le -1\end{cases}$$

9. 编写程序，输入 5 位正整数 M，求出能整除 M 且不是偶数的各个整数，并按从大到小的顺序输出这些整数。
10. 若正整数 M 的全部约数（包括 1 但不包括 M 本身）之和等于 N，且整数 N 的全部约数（包括 1 但不包括 N 本身）之和等于 M，则 M, N 称为亲密数。编写程序计算并输出 10 对亲密数 M, N。
11. 如果一个数等于它的各个因子之和，那么这个数就称为“完数”。例如，6 = 1 + 2 + 3。编写程序，输出 10000 以内的所有完数。

第 5 章　函　数

人类在社会生产活动中遇到的问题通常会超过人类的认识和处理能力。在这种情况下，常用的做法是首先将一个比较复杂的问题分解为一组比较简单的问题，然后分别求解这些简单的问题，最后把简单问题的解组合起来得到复杂问题的解。例如，生产一辆汽车时，首先把汽车分解为发动机、变速箱、传动系统、车体等组成部分，由不同的厂家生产不同的组成部分，然后把它们组装成一辆完整的汽车。

同样，使用计算机程序设计语言编写程序去求解一个问题时，人们也常常遵循这种把复杂问题简单化的思路：首先把一个大问题分解为一系列小问题，分别求解这些小问题，然后把小问题的解合并起来就构成大问题的解。在程序设计中，对大问题的分解过程称为模块化，分解后的一个小问题称为一个模块。

Python 语言用一个函数来实现一个模块，于是一个函数就变成一个模块的解，并且不同的函数之间可以互相调用，通过函数之间的调用把各个模块的解合并起来，进而得到大问题的解。这些函数合并起来后就是一个完整的 Python 语言程序。

Python 语言中的函数是具有名字的能够完成指定任务的语句块。Python 的函数包括内置函数和自定义函数。第 2 章中介绍了 Python 的内置函数，内置函数由 Python 系统提供，是 Python 系统的基本成分，其中的每个内置函数都具有特定的功能。本章首先介绍 Python 用户自定义函数的基本方法、函数的调用方法和函数的参数传递方法，然后简单介绍 4 个比较实用的第三方 Python 程序库的使用方法。

5.1　函数定义

Python 程序开发者可以根据实际问题的需要在程序中定义合适的函数，通常称这类函数为自定义函数。开发者将某项相对独立且可能要多次执行的一组操作语句作为一个整体封装起来，就形成了一个函数。

在 Python 中，函数定义的一般形式如下：

```
def 函数名([形式参数列表]):
    函数体
    [return  值]
```

其中，def 是定义函数的关键字。函数名要符合标识符的命名规则。函数命名要符合见名知义的原则。括号中的内容是函数的参数，由于定义函数时系统还未给这些参数分配存储空间，因此称为形式参数。例如，

```
def  fun(a, b):
    d = a * a - b * b
    return  d
```

定义了一个函数，函数名为 fun，函数有两个参数 a 和 b。函数体中定义了一个变量 d，且变量 d 的值为参数 a 和 b 的平方差，函数体的最后一个语句返回变量 d 的值。

函数可以没有参数，这样的函数称为无参函数。无参函数的定义形式如下：

```
def 函数名():
    函数体
```

定义无参函数时，函数名后面的括号不能省略。例如，

```
def  sayBye():
    print("Good Bye!")
```

定义了一个名为 sayBye 的无参函数，函数体中只有一个语句，它输出一个字符串。

一般情况下，一个函数是被调用来进行信息处理的，函数应把处理的结果返回给调用者，调用者根据该结果就能了解信息在函数内的处理情况。函数通过 return 语句返回值。return 语句的一般形式如下：

```
return 表达式
```

它的作用是返回表达式的值。例如，

```
def sum(a, b):
    return a+b
```

函数 sum()的功能是计算两个数的和，它通过语句 return a+b 计算两个数的和，并且将计算结果返回给调用者。

使用 return 语句时需要注意如下 3 种情况。

（1）return 语句未指定返回值

函数的 return 语句可以不明确指定返回值，函数中甚至可以没有 return 语句。例如，如下定义的函数中，return 后面没有表达式指明其返回值，Python 默认其返回值为 None。

```
def welcome():
    print("Welcome...")
    return
```

（2）return 语句指定返回多个值

函数也可以返回多个值。定义时将需要返回的多个值依次放在 return 的后面，并用逗号隔开，这种形式其实就是返回一个元组。例如，

```
def multi(a, b):
    return  a+b, a-b
```

该函数的 return 语句一次性返回两个表达式的值，它们分别是 a、b 的和与差。

（3）多个 return 语句

一个函数的函数体中可以有多个 return 语句，执行其中任意一个 return 语句后，函数体的执行就结束，即本次函数调用结束并返回调用者。return 语句返回的值就是本次被调用函数的返回值。例如，

```
def  fun(a, b):
    if a < b:
        return b - a
    elif a > b:
        return a - b
    return 0
```

函数 fun(a, b)虽然有三个 return 语句，但根据函数体中各语句的执行逻辑可知有且仅有一个 return 语句被执行，执行到其中一个 return 语句时就会退出该函数，即本次 fun(a, b)函数调用不再执行函数体中的其他语句。

5.2 函数调用与参数

5.2.1 函数调用的一般形式

如果要使用已经定义的某个函数执行指定的任务，那么就要使用函数调用语句来调用它。函数调用需要 Python 系统执行被调用函数的函数体（语句块），如果被调用的函数设置了形式参数，那么函数调用语句必须给每个形式参数指定数据值（这些数据值被称为实际参数，简称实参）。实参可以是具体的数据值（如数字、字符串、列表等），也可以是已经存储具体数据值的变量。函数调用的一般形式如下：

```
函数名([实际参数列表])
```

调用函数时系统已经为括号中的参数分配了存储空间，故称为实际参数。执行函数调用时，系统会将实际参数的值传递给形式参数。例如，

```
def minus(a, b):        # 函数定义
    return a - b
c = minus(8, 3)         # 函数调用
```

函数定义 minus(a, b)中包含形式参数 a 和 b，执行语句 c = minus(8, 3)调用函数时的实际参数为 8 和 3。调用函数的命令执行时，系统将实际参数 8 和 3 分别传递给形式参数 a 和 b。

出现函数调用时，调用语句与被调用函数的执行顺序如下：暂停调用语句转去执行被调用函数，被调用函数执行完后恢复执行调用语句。图 5.1 中给出了调用函数的执行顺序。

例如，函数 callee()的定义如下：

```
def  callee():
    print("被调用函数执行")
```

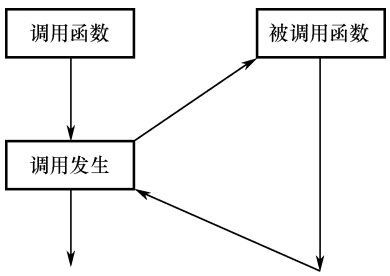


图 5.1　调用函数的执行顺序

主程序如下：

```
print("调用函数之前")
callee()  # 调用函数
print("调用函数之后")
```

执行主程序的输出如下：

```
调用函数之前
被调用函数执行
调用函数之后
```

从这个例子可以看出调用函数和被调用函数的执行顺序。

5.2.2　不可变对象和可变对象参数

Python 中的数据对象分为两种，即不可变对象和可变对象。数字、字符串和元组是不可变对象，列表、集合和字典等是可变对象。

不可变对象作为函数实参时，会在函数体内改变形参的值，但不会导致实参值发生改变。例如，

```
def change(a):
    a = a + 1
    print("a =", a)
b = 4
change(b)
print("b =", b)
```

输出如下：

```
a = 5
b = 4
```

实参 b 的值为 4，调用时，将它的值传递给形参 a。在函数 change()中，将形参 a 的值增加 1，这将只改变形参 a 的值。函数调用结束后，实参 b 的值没有发生改变，仍为 4。

与之相反，可变对象作为函数实参时，会在函数体内改变形参的值，导致实参的值发生改变。例如，

```
def changeList(a):
    for i in range(len(a)):
        a[i] = a[i] * 2
    print("a =", a)
b = [1, 2, 3]
print("b =", b)
changeList(b)
print("b =", b)
```

实参为列表 b，调用函数之前它的值为[1, 2, 3]。调用函数 changeList()时，将它的值传递给形参 a，在函数 changeList()的内部，将形参 a 的每个元素变为原来的 2 倍，形参 a 的值变为[2, 4, 6]。调用结束后实参 b 的值也变为原来的 2 倍。程序的输出如下：

```
b = [1, 2, 3]
a = [2, 4, 6]
b = [2, 4, 6]
```

5.2.3　默认值参数

在 Python 中定义函数时，可以为形式参数设置默认值。给形参设置默认值的一般形式如下：

```
def 函数名(…, 形参名=默认值):
    函数体
```

在定义具有默认值参数的函数时，具有默认值的参数右边不能再出现没有默认值的参数。

调用时可以不给具有默认值的形参传值，这时形参将得到设置的默认值。例如，

```
def defaultPara(a, b=1):
    print("a = {}, b = {}".format(a, b))
```

defaultPara()的参数 b 设置了默认值 1。如果调用时没有为它传值，那么 b 的值为 1。例如，

```
defaultPara(2)     # 调用，b 未传值
```

输出如下：

```
a = 2, b = 1
```

调用时也可以给具有默认值的形参传值，这时形参将得到传递的实参值。例如，

```
defaultPara(2, 3)
```

输出如下：

```
a = 2, b = 3
```

对于设置了默认值的函数，可以使用“函数名.__defaults__”查看函数所有默认值的设置值，返回值是一个元组，其中每个元素依次表示每个默认值参数的设置值。例如，

```
def defaultPara(a = 1, b = 2):
    print("a = {}, b = {}".format(a, b))
x, y = defaultPara.__defaults__
print("a = {}, b = {}".format(x, y))
```

输出如下：

```
a = 1, b = 2
```

设置了默认值的参数只在定义时解释和初始化一次。一个形参如果设置了列表、集合及字典等可变类型的默认值，那么第一次使用默认值时，形参的值为设置的默认值，以后继续调用函数则不会再次用默认值初始化。例如，

```
def defaultPara(a, b = []):
    b.append(a)
    print("b = ", b)
defaultPara(1)
defaultPara(2)
defaultPara(3)
```

函数 defaultPara 的形参 b 设置了一个空列表作为默认值，第一次调用时形参 b 的值为默认值（空列表），第二次调用时 b 的值不再是空列表，而是第一次执行后的列表[1]……三次调用的输出结果如下：

```
b = [1]
b = [1, 2]
b = [1, 2, 3]
```

如果调用时给默认值参数传了值，那么它对默认值形参的变化情况没有影响。例如，

```
def defaultPara(a, b = []):
    b.append(a)
    print("b = ", b)
defaultPara(1, [2])
defaultPara(3)
defaultPara(4, [5])
defaultPara(6)
```

形参 b 的默认值为空列表。第一次调用 defaultPara(1, [2])给形参 b 传递了值[2]，这个值不影响 b 的默认值，因此第二次调用 defaultPara(3)时，形参 b 的值依然是空列表，调用后 b 的值为[3]。第三次调用 defaultPara(4, [5])时，又给形参 b 传递了值[5]，这次调用依然不改变 b 的值，因此第四次调用 defaultPara(6)时形参 b 的值没有改变，还是为[3]。以上 4 次调用输出是：

```
b = [2, 1]
b = [3]
b = [5, 4]
b = [3, 6]
```

5.2.4　关键字参数

通常情况下，实参和形参的结合是按从左到右的顺序一一对应的。通过关键字参数，在调用函数时指定形参的名字，可以改变这个对应关系，实现实参不按照形参的顺序书写。例如，有如下函数定义：

```
def keyPara(a, b, c):
    print("a = {}, b = {}, c = {}".format(a, b,c))
```

如果需要给形参 a、b 和 c 分别传值 1、2 和 3，那么需要按如下方式调用：

```
keyPara(1, 2, 3)
```

即实参和形参的顺序一致。

如果使用关键字，那么可以按如下方式调用：

```
keyPara(c = 3, b = 2, a = 1)
```

即实参和形参的顺序不一致。

这两种调用方式的输出相同：

```
a = 1, b = 2, c =3
```

5.2.5　不定长参数

如果在定义函数时不能确定形式参数的数量，那么需要定义不定长参数。不定长参数的定义形式有两种。第一种定义不定长参数的形式如下：

```
def 函数名(*形参名):
    函数体
```

即在形参名的前面加上一个“*”。系统将传递进来的任意数量的实参组合成一个元组传给形参。例如，

```
def varPara(*p):
    print(p)
varPara(1, 2, 3)
varPara("a", [4, 5], "bc")
```

输出如下：

```
(1, 2, 3)
('a', [4, 5], 'bc')
```

以这种方式定义的函数，实际参数也可以是列表、元组、集合及其他可迭代对象。不论是什么类型的实参，形参的类型都是元组。例如，

```
def varPara(*p):
    print(type(p))
e = {"y":2, "x":1, "z":3 }
a = [1, 2, 3]
b = {1, 2, 3}
c = (1, 2, 3)
varPara(*a)
varPara(*b)
varPara(*c)
varPara(*e.values())
```

以上 4 次调用函数的输出都是

```
<class 'tuple'>
```

第二种定义不定长参数的形式如下：

```
def 函数名(**形参名):
    函数体
```

即在形参名的前面加上两个“**”。系统将传递进来的任意数量的、显式赋值的实参组合成一个字典传给形参。例如，

```
def varPara(**p):
    for it in p.items():
        print(it)
varPara(a = 1, b = 2, c = 3)
```

输出如下：

```
('a', 1)
('b', 2)
('c', 3)
```

以这种方式定义的形参，实参也可以是字典。如果实参是字典，实参名前面也要加上两个星号（**）。例如，

```
e = {"y":2, "x":1, "z":3 }
varPara(**e)
```

5.2.6 实参序列解包

实参序列解包是指当函数具有多个形式参数时，可以使用列表、元组、集合、字典及其他可迭代对象作为实参，并在实参名称前面加上一个星号（*）。如果实参是字典，那么需加两个星号（**），Python 解释器会首先将实参序列解包，然后将序列中的值分别传递给形参变量。例如，有函数定义如下：

```
def varPara(x, y, z):
    print("The sum is {}".format(x+y+z))
```

以下两行代码实现对列表解包：

```
a = [1, 2, 3]
varPara(*a)
```

以下两行代码实现对集合解包：

```
c = {1, 2, 3}
varPara(*c)
```

有两种方法对字典的键进行解包，第一种是

```
d = {1:"r", 2:"s", 3:"t" }
varPara(*d)
```

第二种是

```
d = {1:"r", 2:"s", 3:"t" }
varPara(*d.keys())
```

以下两行代码对字典的值进行解包：

```
e = {"r":1, "s":2, "t":3 }
varPara(*e.values())
```

上面的两种方法通过函数 keys()和 values()将字典的键和值分别转换为可迭代对象，实现了对字典键和值的分别解包。元素加入字典的顺序决定了传递给形参变量的值。例如，函数定义如下：

```
def varPara(x, y, z):
```

```
        print("x = {}, y = {}, z = {}, The sum is {}".format(x, y, z, x+y+z))
```

以下两行代码对字典的值进行解包，传递给形参变量 x、y 和 z 的值分别为 2、1 和 3：

```
e = {"y":2, "x":1, "z":3 }
varPara(*e.values())
```

当字典的键名与形参名一致时，可以按字典键与形参名一一对应的关系进行解包，这时需要使用两个星号（**）的方式。对于前面的一个例子，以下代码传递给形参变量 x、y 和 z 的值分别为 1、2 和 3：

```
varPara(**e)
```

需要注意的是，实现实参序列解包时要求实参对象的数量和形参的数量一致。

5.3　变量的作用域

变量的作用域是指变量起作用的范围，它由定义变量的位置决定。不同作用域内的变量之间互不影响。根据变量定义的位置，变量分为局部变量和全局变量。

局部变量是指在函数内部定义的变量，它只在定义该变量的函数内部起作用，作用范围从定义该变量的位置开始到函数结束。例如，

```
def  local():
    a = 3
print(a)      # 将报错
```

在函数 local()中定义的变量 a 是局部变量，它的作用域只限于该函数内部，函数 local()外部的语句 print(a)试图访问 a 时将会报错。

形参变量也是局部变量，它的作用范围也只限于该函数内部，因此实参变量和形参变量可以同名。例如，

```
def local(b):
    a = 3
    print("a = {}, b = {}".format(a, b))
b = 4
local(b)
```

在 if 语句和循环语句的语句块中定义的变量及 for 循环的循环变量都是局部变量，它们的作用范围都持续到函数结束。例如，

```
def local(a):
    if a > 0:
        b = 3
    for c in range(5):
        c = 4
        d = 5
    print("a = {}, c = {}, d = {}".format(b, c, d))
```

全局变量是指在函数外部定义的变量，其作用范围从定义该变量的位置开始到程序结束。如果在函数内只需要引用全局变量，那么可以直接用它给其他变量赋值或输出它的值。例如，

```
a = 5              # 全局变量
def glob():
    b = a          # 使用了全局变量 a
    print("a = {}, b = {}".format(a, b))
```

```
glob()
```

变量 a 在函数外部定义，在函数体内可以用来给局部变量 b 赋值，也可以输出它的值。

如果要在函数体内部修改全局变量 a 的值，那么要用关键字 global 声明变量 a。例如，

```
a = 5
def glob():
    global a      # 声明此处的 a 是全局变量
    a = 4
glob()
print("a = {}".format(a))
```

在给 a 赋值 4 之前，首先用 global a 声明，然后就可把全局变量 a 修改为 4，最后的输出为 a = 4。

局部变量和全局变量可以同名。在这种情况下，在函数体内部给该变量赋值时，全局变量不起作用，只是改变局部变量的值。例如，

```
a = 5             # 全局变量
def glob():
    a = 4         # 此处 a 是局部变量
glob()
print("a = {}".format(a))
```

首先定义了一个全局变量 a，并赋值 5；然后定义了一个局部变量 a，并赋值 4。这样做只改变了局部变量 a 的值，全局变量 a 的值仍然为 5。

5.4 lambda 表达式

lambda 表达式常用来定义匿名函数。有时我们需要使用一个函数，但只使用这个函数一次。例如，内置函数 filter()和 map()的第一个参数，内置函数 sorted()和列表的 sort 方法的 key 参数等。在这种场合下，比较适合用 lambda 定义一个匿名函数。lambda 表达式的一般形式如下：

```
lambda 形参列表: 表达式
```

关键字 lambda 与形参列表之间用空格分隔。冒号后面为匿名函数的函数体，只能包含一个表达式，表达式的值就是该匿名函数的返回值。例如，

```
from random import randint
a = [randint(-100,100) for i in range(10)]    # 产生一个含正数和负数的随机整数列表
print('排序前:', a)
a.sort(key = lambda x : x * x)                # 根据数据的平方值排序
print('排序后:', a)
```

以上程序段的功能是随机产生一个列表，并对该列表按元素平方值从小到大的顺序排序，分别输出排序前和排序后的列表。随机执行的可能输出如下：

```
排序前: [11, -16, 66, -34, -10, -88, -23, 0, -94, 40]
排序后: [0, -10, 11, -16, -23, -34, 40, 66, -88, -94]
```

排序用到列表的 sort 函数()，该函数的参数 key 是一个函数，它指定排序规则。如果不使用 lambda 表达式实现该功能，那么需要额外定义一个函数，具体实现如下：

```
from random import randint
def st(x):                        # 先定义一个函数
    return x * x
```

```
a = [randint(-100,100) for i in range(10)]
print('排序前:', a)
a.sort(key = st)                # 指定用 st 函数排序
print('排序后:', a)
```

比较两种方法，我们发现使用 lambda 表达式更加简洁。

也可以将 lambda 表达式变成命名函数，一般形式如下：

```
函数名 = lambda 形参列表: 表达式
```

例如，

```
f = lambda x, y: x + y
```

定义了一个命名函数 f，其形式参数为 x 和 y，返回值为 x + y。对该函数的调用方式为 f(x, y)。例如，

```
print(f(2, 3))
```

将 lambda 表达式变成命名函数后，它也支持函数的一些特性，如默认参数、关键字参数等。例如，

```
g = lambda a, b = 3: a*a + b*b
```

以下语句使用了默认参数：

```
print(g(2))
```

以下语句使用了关键字参数：

```
print(g(b = 4, a = 3))
```

对 lambda 表达式的使用建议是：尽量用它来完成简单的任务。如果用它来完成一些复杂的功能，那么出错的概率就会增加。例如，

```
a = []
for i in range(5):
    a.append(lambda: i * 2)
for j in range(5):
    print(a[j](), end = ' ')
```

以上程序段的输出是 8 8 8 8 8，而不是期望的 0 2 4 6 8。出现这个错误的原因是添加到列表 a 中的元素都是表达式 lambda: i * 2，这个表达式的值只取决于变量 i 的值，而不取决于列表 a 的下标。当第一个 for 循环结束后变量 i 的值为 4，所以在第二个 for 循环的输出语句中，随着下标 j 的变化，a[j]()访问了列表 a 的不同元素，它们的值是相同的，即都是 8。

只有当 i 的值发生改变后，输出结果才会发生改变。例如，

```
i = 3
for j in range(5):
    print(a[j](), end = ' ')
```

输出结果为 6 6 6 6 6。虽然输出值发生了改变，但它们依然与列表 a 的下标无关。

如果把下标改为 i，例如，

```
for i in range(5):
    print(a[i](), end = ' ')
```

那么输出结果为 0 2 4 6 8。因为下标与变量 i 相同，所以改变下标的同时也改变了变量 i 的值。

5.5　嵌套定义、修饰器和生成器函数

Python 函数可以嵌套定义，即在函数体内再定义另一个函数。嵌套函数的定义形式与一般函数的一样，只是要位于一个函数的函数体内。例如，

```
def outer():
    def inner():
        print("inner function")
    print("outer function")
```

以上代码定义了一个函数 outer()，在它的函数体内又定义了一个函数 inner。

被嵌套的函数不会随着外部函数的执行而自动执行。例如，有如下语句：

```
outer()
```

它的输出是 outer function。

要执行被嵌套的函数，需要在外部函数的函数体内显式地调用它。例如，

```
def outer():
    def inner():
        print("inner function ")
    inner()
    print("outer function ")
```

在外部函数的函数体内有一个语句 inner()调用了被嵌套的函数。有如下语句：

```
outer()
```

它的输出是：

```
inner function
outer function
```

嵌套函数的一个重要应用是修饰器。修饰器也是一个函数，它接收其他函数作为参数，在此基础上加上一些功能作为修饰后返回新函数。例如，下面的代码定义了一个修饰器，它接收函数 fun()作为参数，在它之前加上功能语句 print("前修饰操作")、在它之后加上功能语句 print("后修饰操作")作为修饰。以后，在执行函数 fun()之前都要先执行前修饰功能语句 print("前修饰操作")，在执行函数 fun()之后都要执行后修饰功能语句 print("后修饰操作")。

```
def decorator(fun):
    def wrapper(*args, **kwargs):
        print("前修饰操作")
        result = fun(*args, **kwargs)
        print("后修饰操作")
        return result
    return wrapper
```

要使用该修饰器，需要在定义函数之前加上修饰标识。例如，

```
@decorator              # 修饰标识
def main():
    print("主要功能")
```

在定义函数 main()之前加上@ decorator 以使用该修饰器。调用函数 main()的方法与原来一样。例如，

```
main()
```

输出变为

```
前修饰操作
主要功能
后修饰操作
```

这个例子中的修饰器既有执行函数之前的修饰功能，又有执行函数之后的修饰功能。也可以只有执行函数之前的修饰功能，或者执行函数之后的修饰功能。修饰器函数在开发大型软件或框架时非常有用。

生成器函数是指一个包含了 yield 语句的函数，生成器函数可以用来创建生成器对象。yield 语句和 return 语句都是从函数中返回值。return 语句返回值并结束函数的运行，yield 语句返回值并暂停函数的运行，下次通过生成器对象的__next__()函数、内置函数 next 或 for 循环遍历生成器对象时再恢复执行，并从 yield 语句的后一条语句继续执行。例如，有生成器函数定义如下：

```
def g():
    a = 1
    while True:
        yield  a  # 利用 yield 关键字定义生成器函数
        a = a + 2
```

生成器函数为 g()，c = g()定义一个生成器对象。下面的语句通过内置函数 next()输出前 20 个奇数：

```
c = g()
for i in range(20):
    print(next(c), end = ' ')
```

下面的语句通过 for 循环遍历生成器对象输出小于 40 的所有奇数：

```
d = g()
for  i  in  d:
    if  i >= 40:
        break
    print(i, end = ' ')
```

与普通函数相比，生成器函数可以有效地节省内存。例如，要处理 10 万个数据时，普通函数需要把全部数据一次性生成并返回给主程序，占用的内存较大。生成器函数可以每次只产生一个数据，处理后再产生下一个数据，因此消耗的内存就很小。Python 的 range()函数是一个高效的生成器函数。

5.6　函数递归调用

一个函数可以直接或间接地调用该函数本身，这种调用称为递归调用。

直接递归调用就是一个函数直接调用该函数自身。例如，

```
def f():
    …
    f()
    …
```

直接递归函数调用的调用过程如图 5.2 所示。如果不对函数 f 的运行施加一定的限制条件，那么这个递归调用的过程将变成一个无限的过程，这显然违反了算法必须在有限步内完成的要求。使用递归调用最重要的规则是要设定适当的结束条件，使函数能够终止递归，避免出现无限调用的情形。

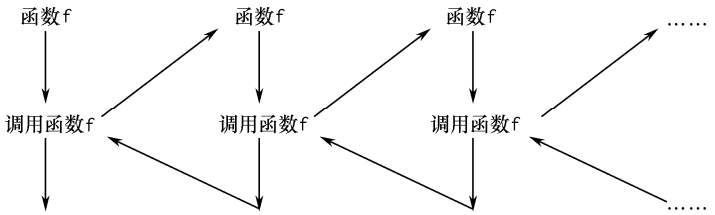


图 5.2 直接递归函数调用的调用过程

间接递归调用是一个函数调用另一个函数，该函数又要调用原函数。例如，

```
def f():
    …
    g()
    …
def g():
    …
    f()
    …
```

间接递归函数调用的调用过程如图 5.3 所示。如果不加限制，那么间接递归调用的过程同样会出现无限调用的情形。使用间接递归函数调用时也要注意设置条件使调用过程在有限步内终止。

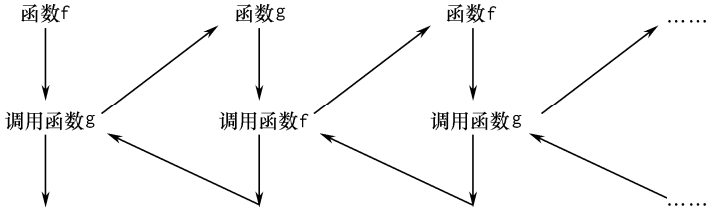


图 5.3 间接递归函数调用的调用过程

求解一个问题时，把该问题分解为两个小问题，小问题与原问题有相似的结构，因此小问题与原问题可以用相同的方法来求解，这样的问题适合用递归函数调用来求解。

把大问题分解为两个小问题，把小问题分解为更小的问题……问题的规模就变得越来越小。分解到某个程度时，问题的规模就会小到很容易求解，这时就不需再进行分解，于是自然地终止了递归调用的过程，避免了无限递归调用的情形。

【例 5.1】编写程序，输入一个自然数 n，求它的阶乘（$n!$），然后输出所求的值。

分析：因为 $n!=n\times(n-1)!$，于是把求自然数 n 的阶乘的问题分解为两个小问题：求 n 与 $n-1$ 的阶乘 $(n-1)!$。第一个问题是求 n 的值，这个问题比较简单，可以直接得到结果；第二个问题是求 $n-1$ 的阶乘 $(n-1)!$，它与原问题求 n 的阶乘 $n!$有着相似的结构，可以用相同的方法求解。所以这个问题可以用递归调用的方法求解。n = 1 时，很容易知道它的阶乘，这时不需要再分解该问题，自然就结束了递归的过程。

递归函数求阶乘的程序如下：

```
def  f(n):
    if  n == 1:              # n=1 时结束递归，返回具体的值
        return 1
    return n * f(n - 1)      # 递归调用 n! = n * (n-1)!
```

```
n = eval(input("Input an integer:"))
fac = f(n)
print("{}! = {}".format(n, fac))
```

【例 5.2】编写递归函数 int fun(int n)，把输入的一个十进制整数转换为二进制数输出。在函数 main()中输入正整数 x，调用函数 fun()，最后输出转换结果。

分析：设整数 n 的二进制数为 M，n/2 的二进制数为 N，则 M=N*2+n%2。这样，就将求整数 n 的二进制数的问题分解为两个问题：求 n/2 的二进制数和求 n%2。第二个问题比较简单，可以直接得到结果。第一个问题（求 n/2 的二进制数）与原问题有着相似的结构，即求一个整数的二进制数只是将整数变为原来的一半。因此，这个问题适合用递归调用的方法求解。

程序如下：

```
def  f(n):
    if  n == 0:             # 当 n 为 0 或 1 时，无须再递归，可返回'0'或'1'
        return  '0'
    elif  n==1:
        return  '1'
    t = n % 2
    n = n // 2
    return  f(n) + str(t)   # 递归调用
n = eval(input("Input an integer:"))
fac = f(n)
print("{} = {}".format(n, fac))
```

5.7　Python 的第三方库

本节介绍 Python 程序设计二级考试要求的几个第三方库，以便掌握二级考试要求的考核内容。

5.7.1　pyinstaller 库

pyinstaller 库的作用是将 Python 程序转换为可执行程序。Python 不包含 pyinstaller 库，需要先安装后才能使用这个库。在线安装命令如下：

```
pip install pyinstaller
```

成功安装后，会在 Python 的安装目录下的 Scripts 子目录中生成一个程序 pyinstaller.exe，使用它就可将 Python 程序转换为可执行程序。pyinstaller 工具是通过命令行窗口使用的，其语法格式如下：

```
pyinstaller  [选项]  python 源文件
```

一个 Python 项目包含多个源文件时，只需在 pyinstaller 命令行中指定包含程序入口的源文件。例如，若源文件为 target.py，则将该程序转换为可执行程序的命令如下：

```
pyinstaller -F -c target.py
```

-F 选项指定生成单独的 EXE 文件，-c 选项指定使用命令行窗口运行程序。如果用户编写的 Python 程序含有图形用户界面（例如使用了 tkinter 库），那么转换命令如下：

```
pyinstaller -F -w target.py
```

-w 选项指定使用图形窗口运行程序。

命令执行后，将在 target.py 所在的目录下增加一个 dist 目录，并在该目录下生成一个 target.exe

文件，双击该文件即可执行程序。将文件复制到未安装 Python 的机器中，双击程序也可运行，因为 target.exe 中已包含了 Python 运行的支持库。

5.7.2 jieba 库

假设在一段中文中，文字与文字之间没有分割符。要研究这段中文中的词语，就要提取出那些要研究的词语。jieba 库的作用就是对中文文章进行分词，提取中文文章中的词语。在线安装命令如下：

```
pip  install  jieba
```

jieba 库中常用的两个函数是 cut()和 lcut()，二者的区别如下：cut()函数的返回值是一个可迭代的生成器对象，lcut()函数的返回值是一个列表对象。以下介绍 cut()函数。cut()函数的使用格式如下：

```
cut(字符串, cut_all, HMM)
```

其中字符串是要进行分词的字符串对象；cut_all 参数为真表示采用全模式分词，为假表示采用精确模式分词，默认值为假；HMM 为真表示采用 HMM 模型，为假则不采用，默认值为真。

在全模式下，字符串中的所有可能词语都会被提取出来。在精确模式下，只提取字符串中最长的一个词语。例如，对于字符串“我在北京大学”，采用精确模式进行分词时的使用方法如下：

```
In: import jieba
In: obj = jieba.cut("我在北京大学", cut_all=False) # cut 返回分词迭代对象
In: list(obj)
Out: ['我', '在', '北京大学']
```

采用全模式进行分词时的使用方法如下：

```
In: jieba.lcut("我在北京大学", cut_all=True)        # lcut 直接返回分词列表
Out: ['我', '在', '北京', '北京大学', '大学']       # 全模式返回所有可能的分词结果
```

HMM 模型，即隐马尔可夫模型（Hidden Markov Model, HMM），是一种基于概率的统计分析模型，用来描述一个系统隐性状态的转移和表现的概率。jieba 库使用该模型来处理未登录到词库的词。例如，对于字符串“我在广石上大学”，不采用 HMM 模型进行分词的使用方法如下：

```
In:jieba.lcut("我在广石上大学", HMM=False)
Out:['我', '在', '广', '石', '上', '大学']
```

采用 HMM 模型进行分词的使用方法如下：

```
In: jieba.lcut("我在广石上大学", HMM=True)
Out: ['我', '在', '广石', '上', '大学']
```

【例 5.3】设有字符串 str。编写程序采用精确模式和 HMM 模型对该字符串进行分词，然后输出每个词及其在字符串中出现的频率，每个词只输出一次。

程序如下：

```
import  jieba
str = "编程语言可分为低级语言和高级语言。"
lst = jieba.lcut(str, HMM=True)
b = []
for v in lst:
    if v not in b:
        b.append(v)
        print(v, "\t", lst.count(v))
```

5.7.3　wordcloud 库

wordcloud 库的功能是将一组给定的词语按照词频转换为一张图片，其中高频词显示的占比较大，从而突出重点词汇。在线安装命令如下：

```
pip install wordcloud
```

使用 wordcloud 库时，首先需构造一个 wordcloud 库的 WordCould 对象，然后调用该对象的 generate()函数生成词云，最后调用该对象的 to_file()函数保存生成的图片。构造 WordCloud 对象的格式如下：

```
wc = WordCloud(width = 1000, height = 800, background_color='white')
```

其中，参数 width 指定生成词云的宽度（单位为像素），参数 height 指定高度（单位为像素），background_color 指定词云的背景颜色。

WordCould 对象的 generate 函数()的使用格式如下：

```
generate(字符串)
```

其参数是用空格分隔的字符串。

例如，有以下字符串：

```
string = 'With the improving of semiconductor technology, a single chip integrates more and more processing cores. Highly parallel applications are distributed to tens of processing units. The inter-processor communication delay becomes more and more important for parallel applications.'
```

下面的代码将生成词云并保存为 wc.png 图片文件：

```
from  wordcloud  import  WordCloud
wc = WordCloud(background_color='white', width=1000, height=800).generate(string)
wc.to_file('wc.png')
```

生成的词云如图 5.4 所示。

图 5.4　英文字符串词云图

如果是中文文章，那么需要首先对文章进行分词，并用空格连接分词的结果，然后才能用 WordCloud 生成词云。例如对于以下中文字符串：

```
string = '编写程序就是用计算机语言实现算法的过程。可以证明，任何问题都可以由三种基本
         结构表达出来。这三种基本结构包括顺序结构、选择结构和循环结构。'
jbstr = ' '.join(jieba.lcut(string))
wc = WordCloud(font_path='simkai.ttf', background_color='white',
     width=500, height=400)  # 楷体
wc.generate(jbstr) # 生成词云
wc.to_file('wc.png')    # 保存为图片文件 wc.png
```

上面的代码中，先调用 jieba 库对中文字符串进行分词处理，并用空格连接提取出来的词语，得到一个用空格分隔的字符串。在用中文字符串生成词云时，需要给 WordCloud 对象的构造函数指定中文字体，即第一个参数 font_path。生成的词云如图 5.5 所示。

图 5.5　中文字符串词云图

5.7.4　turtle 库

turtle 库是 Python 内包含的一个简易绘图库，它通过模拟一只乌龟在画板上的爬行来绘制图形。画板采用笛卡儿坐标系，原点(0, 0)位于画板中央。最初，乌龟位于坐标系的原点，朝向 X 轴的正方向。只需设定乌龟爬行的方向和距离，就能沿指定的方向绘制一条直线或曲线。turtle 库的距离参数单位为像素。turtle 库中的常用函数如下：

1. forward()或 fd()：向前移动指定的距离，参数为数值，单位像素。例如，turtle.forward(25)向前绘制一条长度为 25 像素的直线。
2. backward()或 bk()：向后移动指定的距离，参数为数值。例如，turtle.backward(30)向后绘制一条长度为 30 像素的直线。
3. right()或 rt()：以角度单位向右转动。参数为一个数值，单位默认为度，可以通过函数 degrees()和 radians()进行角度或弧度变换。例如，turtle.right(45) 控制乌龟画笔向右转动 45 度。下面的程序段将在画板上画一个边长为 200 像素的矩形，如图 5.6 所示。

```
import turtle
for  _  in  range(4):
    turtle.fd(200)      # 绘制 200 像素长的直线
    turtle.right(90)    # 每绘一根线条就右转 90 度
turtle.done()           # 绘图结束（在 Spyder 中调试 turtle 程序时，程序末尾需要此命令）
```

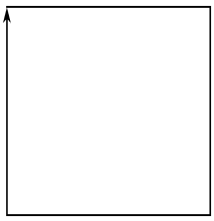


图 5.6　边长为 200 像素的矩形

4. left()或 lt()：以角度单位向左转动，参数为一个数值。例如，turtle.left(45)控制乌龟画笔向左转动 45 度。

5. goto()或 setposition()：移到绝对坐标位置。此时画笔落下并移动时将画线，不改变方向。包含两个数值参数 x 和 y。例如，turtle.goto(60,30)将绘图乌龟移到坐标(60,30)处。
6. setx()：设置乌龟沿 X 轴方向移动的距离，参数为数值。例如，turtle.setx(10)使乌龟沿 X 轴方向画一条长为 10 像素的直线。
7. sety()：设置乌龟沿 Y 轴方向移动的距离，参数为数值。例如，turtle.sety(50)使乌龟沿 Y 轴方向画一条长为 50 像素的直线。
8. setheading()或 seth()：设置乌龟前进的方向，参数为数值。各坐标轴的方向值为：参数值为 0 表示 X 轴正方向，即向东；参数值为 90 表示 Y 轴正方向，即向北；参数值为 180 表示 X 轴负方向，即向西；参数值为 270 表示 Y 轴负方向，即向南。例如，turtle.seth(90)设置乌龟前进的方向为向北。
9. home()：将乌龟移回原点(0,0)，并将方向设置为初始的正东方向，无参数。
10. circle()：从当前点出发绘制一个给定半径的圆或圆弧。例如，turtle.circle(100)绘制一个半径为 100 像素的圆，turtle.circle(100, 80)绘制半径为 100、圆心角为 80 度的圆弧。半径为正数时，逆时针方向绘图，圆心在前进方向的左侧。turtle.circle(-200, 120)绘制半径为 200、圆心角为 120 度的圆弧。半径为负数时，顺时针方向绘图，圆心在前进方向的右侧。
11. dot()：以当前点为圆心画一个给定直径的圆点。第一个参数是一个大于 1 的整数，默认值为 pensize+4 和 2*pensize 的最大值；第二个参数设置原点的颜色值。例如，turtle.dot(20, "blue")画一个直径为 20 的蓝色圆点。
12. penup()：抬起画笔，这样移动笔时就不会画出线条。
13. pendown()：放下画笔，这样绘图时将画出线条。
14. pencolor()：设置笔的颜色，参数为一个表示颜色的字符串。例如，pencolor("red")将笔的颜色设为红色。
15. pensize()：设置画笔的宽度，参数为一个数值。例如，pensize(10)将画笔宽度设为 10。
16. speed()：设置绘图速度，参数可以是诸如"slow"或"fast"这样的字符串，也可以是 0～10 之间的一个整数，区间[1, 9]内的数值越大，绘图速度越快，speed(0)速度最慢。
17. done()：表示绘图结束。在 Spyder 中运行 turtle 程序时，程序末尾应包含此命令。

下面的代码绘制一个正六边形，如图 5.7 所示。

```
import turtle as t
for i in range(6):
   t.fd(100)
   t.seth(60 + 60 * i)
t.done()
```

下面的代码绘制了一个太阳花图案，如图 5.8 所示。

```
import turtle as t
t.color("red", "yellow")    # 设置画笔颜色和填充颜色
t.begin_fill()              # 开始填充
for _  in  range(50):
    t.forward(200)
    t.left(170)
t.end_fill()                # 结束填充
t.done()
```

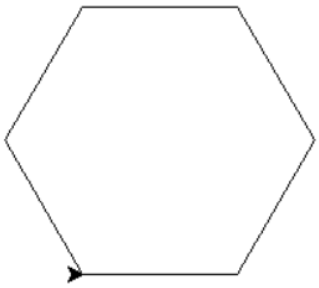


图 5.7　正六边形

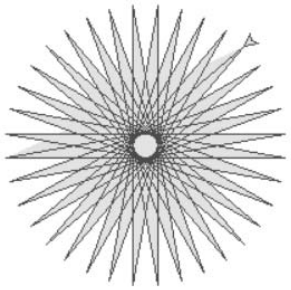


图 5.8　太阳花图案

5.8　本章小结

本章介绍了 Python 函数的使用。首先介绍了函数的定义；接着介绍了函数调用和函数参数，重点介绍了使用函数参数需要注意的几种情况（不可变对象和可变对象、默认值参数、关键字参数、不定长参数、实参序列解包等）；然后介绍了变量作用域的相关问题、实现匿名函数的 lambda 表达式、函数的嵌套定义、修饰器和生成器函数及函数的递归调用；最后介绍了二级考试涵盖的 pyinstaller 库、jieba 库、wordcloud 库及 turtle 库。

习题

1. 编写一个函数 fun(a)，该函数只有一个参数 a，判断参数 a 的类型，若 a 是组合数据类型则输出“组合数据类型”，否则输出“非组合数据类型”。
2. 编写函数 fun(n)，其功能是计算并输出多项式 s=1+1/(1+2)+1/(1+2+3)+…+1/(1+2+3…+n)的值。在主程序中输入正整数 n，调用函数 fun，最后输出计算结果。
3. 编写函数 fun(x)，其参数是一个元素为实数的列表。计算并输出给定列表中每相邻两个元素的平方根之和。在主程序中输入列表的值，调用函数 fun，并输出计算结果。
4. 编写一个 lambda 表达式，调用内置函数 sorted 对列表[111，55，4]进行排序，按列表元素对应字符串的长度从小到大排序。
5. 编写函数，接收任意数量的整数作为参数，返回一个元组，元组的第一个元素是所有参数的中位数，第二个元素是所有小于中位数参数的平均值，第三个元素是所有大于中位数参数的平均值。
6. 编写一个函数，其参数为两个正整数，将这两个正整数之间的所有素数以一个元组的形式返回。
7. 定义一个生成器对象，生成 100 以内的所有偶数。
8. 有 5 个人坐在一起，问第五个人多少岁，他说比第四个人大 2 岁。问第四个人多少岁，他说比第三个人大 2 岁。问第三个人，他说比第二个人大 2 岁。问第二个人，他说比第一个人大 2 岁。最后问第一个人，他说是 10 岁。请问第五个人多大？编写递归函数 fun(n)，求第五个人多大。在函数 main 中调用函数 fun，并输出计算结果。

第 6 章　文　件

程序运行时，用变量或组合数据对象保存需要处理的数据，变量和数据对象都存储在内存中。内存中的数据不能永久保存，一旦断电，这些数据就会丢失。如果要永久保存内存中的数据，那么就要将数据保存到外部存储器中，以后需要时再从外部存储器中读取。本章介绍如何将数据保存到外部存储器中及如何从外部存储器中读取数据。

6.1　文件的基本概念

文件是指存储在外部存储器中的一组信息集合。操作系统以文件为单位对外部存储器中的数据进行管理，操作系统用文件名对文件进行标识。要向外部存储器中存储数据，必须先建立一个文件，然后把数据保存到该文件中。反之，要从外部存储器中读取数据，也要首先指定保存数据的文件名，打开该文件，然后才能通过该文件读取存储的数据。

按照文件的数据组织形式，文件分为文本文件和二进制文件两种。文本文件将处理的数据视为一个字符串，把字符串中每个字符的编码保存到文件中。例如，要存储字符串“world”，可将该字符串中每个字符的编码依次保存到文件中，保存结果如图 6.1 所示。

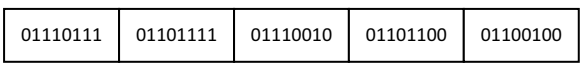


图 6.1　文本文件存储图

二进制文件把数据的二进制值存储到文件中。例如，有一个整数 20190306，因为一个整数占 4 字节，所以把它保存在外部存储器中也要占 4 字节。将这个整数以二进制形式保存到文件中的结果如图 6.2 所示。

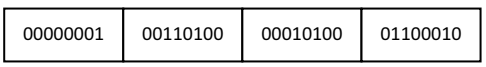


图 6.2　二进制文件存储图

将数据保存到外部存储器（外存）中后，程序要使用数据时，需要从外存内的文件中读取。程序从外存中读取数据的方式有非缓冲方式和缓冲方式两种。使用非缓冲方式时，直接把数据从外存读到程序的数据区。使用非缓冲方式读取数据的速度往往较慢，使用缓冲区可以提高读取数据的速度。缓冲区是内存中的一段存储区域，存取文件时，系统会自动地为每个文件开辟一个缓冲区。如果要从外存读取数据，那么一次将一批数据读入缓冲区，程序运行需要数据时，直接从缓冲区中读取。如果要向外存中写入数据，那么首先将数据保存到缓冲区中，然后一次性地将缓冲区中的数据全部写到外存中。文件缓冲区示意图如图 6.3 所示。

Python 读/写文件的基本流程如图 6.4 所示。在对文件进行读/写操作之前，先要打开文件，打开成功才能读/写文件内容，读/写完毕还要关闭文件。若打开文件失败，则不能对文件进行读/写操作，这时需要检查打开失败的原因。

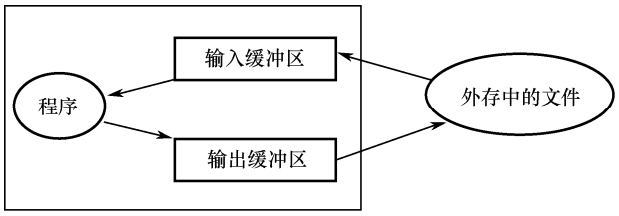


图 6.3 文件缓冲区示意图

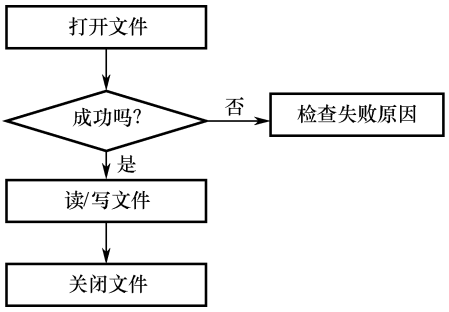


图 6.4 读/写文件的基本流程

6.2 文件基本操作

6.2.1 用内置函数 open 打开文件

在 Python 中，用内置函数 open()打开文件。使用 open 打开文件的一般形式如下：

```
f = open(文件名，文件使用模式，编码方式，缓冲区大小)
```

其中，文件名是指定要打开的文件的名称。文件使用模式也是一个字符串，它指定文件的使用模式，默认值为“r”。编码方式用来指定文件中字符的编码，默认编码与操作系统有关。缓冲区大小是一个整数，它指定文件缓冲区的大小：0 表示没有缓冲区，负数表示使用系统默认的缓冲区大小，默认值为-1。函数 open()的返回值是一个文件对象。

Python 中文件的使用模式如表 6.1 所示。

表 6.1 文件的使用模式

文件使用模式	含　义
"r"	以只读方式打开一个文件
"w"	以只写方式打开一个文件，若文件已存在，则先清空原有内容
"a"	以向文件末尾追加数据的方式打开一个文件，不覆盖文件的原有内容
"x"	以写模式新建一个文件，若文件已存在，则抛出异常
"b"	打开一个二进制文件，可与其他模式组合使用
"t"	打开一个文本文件，可与其他模式组合使用
"+"	以读写方式打开一个文件，可与其他模式组合使用

打开文件时需要指定打开的是文本文件还是二进制文件（"t"或"b"），默认为"t"。例如，

```
fw = open(r"d:\test.txt", "w")
```

该语句的作用是以“只写”方式打开 d 盘根目录下的文本文件"test.txt"，若成功则返回文件对象 fw。

```
fr = open(r"d:\file1.txt", "r")
```

该语句的作用是以“只读”方式打开 d 盘根目录下的文本文件"file1.txt"，若成功则返回文件对象 fr。

```
fbw = open(r"d:\file2.dat", "wb")
```

该语句的作用是以“只写”方式打开 d 盘根目录下的二进制文件"file2.dat"，若成功则返回文件对象 fbw。

```
fw = open(r"d:\test.txt", "r", encoding="utf-8")
```

该语句的作用是以“只读”方式打开 d 盘根目录下的文本文件"test.txt"，指定编码方式为"utf-8"，若成功则返回文件对象 fw。由于该文件保存时采用的是"utf-8"编码，因此打开该文件时需要指定这种编码方式。如果不指定，或指定为其他编码方式，那么不能打开该文件。

6.2.2　文件对象的属性和常用方法

函数 open()返回的是一个可迭代的文件对象，通过该对象可以对文件进行读/写操作。文件对象的常用属性如表 6.2 所示。

表 6.2　文件对象的常用属性

属　性	类　型	说　明
closed	布尔型	判断文件是否关闭，关闭为 True，否则为 False
mode	字符串	文件的打开模式
name	字符串	文件的名称

文件对象的常用方法如表 6.3 所示。

表 6.3　文件对象的常用方法

方　法	功　能	返 回 值
close()	把缓冲区的内容写入文件，关闭文件并释放缓冲区	无
flush()	把缓冲区的内容写入文件，不关闭文件	无
read([size])	从文件读取指定的字符数，若未给定或为负则读取所有内容	从文件中读取的字符
readline()	从文件中读取一行，包括"\n"字符	从文件中读取的一行字符串
readlines()	读取所有行（直到结束符 EOF）并返回列表	字符串列表，包含所有的行
seek(offset [,whence])	移动文件指针。offset：开始的偏移量，代表需要偏移的字节数；whence：可选，默认值为 0，0 代表从文件开头算起，1 代表从当前位置算起，2 代表从文件末尾算起	无
tell()	返回文件的当前位置，即文件指针的当前位置	文件的当前位置
write(s)	向文件中写入指定字符串	写入的字符长度
writelines(s)	向文件中写入一个字符串序列	无

6.2.3　关闭文件

完成对某个文件的操作后，一定要关闭这个文件对象，才能将所做的修改保存到文件中。例如，

```
fw.close()
fr.close()
```

在操作文件和关闭文件之前发生错误而导致程序崩溃时，无法正常关闭文件。使用关键字 with 可以避免这个问题。关键字 with 可以自动管理资源，不论什么原因跳出 with 块，总能保证正确关闭文件。关键字 with 的一般形式如下：

```
with  open(文件名, 文件使用模式) as fp:
```

它通过文件对象 fp 读/写文件的语句内容。

6.2.4 读/写文本文件

用文件对象的方法 read()按字符数读文件，换行符"\n"也算一个字符。

【例 6.1】当前文件夹下有一个文件 log.txt，其内容如下：

```
第一行
第二行
```

编写程序读取其中的前 6 个字符。

程序如下：

```
with open("./log.txt", "r") as f:
    a = f.read(6)
    print(a, end= '')
```

输出如下：

```
第一行
第二
```

因为换行符"\n"也算一个字符，所以从该文件读取 6 个字符时，第二行只读了 2 个字符。

【例 6.2】按行遍历文件的所有行。

程序如下：

```
with open("./log.txt", "r") as f:
    a = f.readlines()
    for line in a:
        print(line)
```

输出结果如下：

```
第一行


第二行
```

因为函数 readlines()读取的行也包含换行符"\n"，函数 print()输出时会加一个换行，所以第一行和第二行之间有一空行。为了避免出现空行，只需把输出语句改为

```
print(line, end = '')
```

因为函数 open()返回的文件对象是一个可迭代对象，所以用函数 readlines()读取文件的语句可以省略。该段程序可以改为

```
with open("./log.txt", "r") as f:
    for line in f:
        print(line, end = '')
```

要将数字（整数或实数）写到文本文件中，需要首先将其转换为字符串，然后调用文件对象的函数 write()写入文本文件。函数 write()写入一个字符串时不会自动换行，需要加上换行符。

【例 6.3】随机生成一个长度为 100 的整数列表，其元素范围为 1～100，将该列表以每 10 个一行（元素之间以空格分隔）写入一个文本文件（"d:\record.txt"）。

程序如下：

```
from random import randint
a = []
for i in range(100):
    a.append(randint(1,100))
with open("d:\\record.txt", "w") as f:
```

```
        str = ""
        for i, v in enumerate(a):
            str = str + "{} ".format(v)
            if (i + 1) % 10 == 0:
                b = f.write(str + "\n")
                str = ""
```

当将文本文件中的数字读取到程序并保存到列表中，对文件进行遍历时，函数 readlines()读取的文件行是一个字符串，在把该字符串转换为数字时，需要调用函数 strip()删除行尾的换行符"\n"。

【例 6.4】将文本文件（"d:\record.txt"）中的数字读入一个列表，并输出该列表。

程序如下：

```
b = [ ]
with open("d:\\record.txt") as f:
    for line in f.readlines():
        line = line.strip()
        data = line.split()
        for v in data:
            b.append(eval(v))
print(b)
```

6.2.5　读/写二进制文件

把内存中的数据对象写入二进制文件的过程称为序列化。读取二进制文件中的数据并重建原来的数据对象的过程称为反序列化。本节介绍如何使用 pickle 模块读/写二进制文件。写二进制文件时使用 pickle 模块的 dump()函数，它的一般使用形式如下：

```
dump(写入对象, 文件对象, [,协议])
```

其中：写入对象是要写入文件的对象，它可以是整数、实数、字符串、列表、字典等对象。文件对象是函数 open()打开的文件对象，对象写入其中。协议是序列化使用的协议；若该项省略，则默认为 0；若为负值或 HIGHEST_PROTOCOL，则使用最高的协议版本。

【例 6.5】程序运行时有如下数据对象：

```
a = 1234
b = 3.14159
c = "程序"
d = ['a', 'b', 'c']
e = {"张":60, "王":70, "李":80}
```

编写程序，将它们写入当前目录下的二进制文件 binary.dat。

程序如下：

```
import  pickle
with open("binary.dat", "wb") as f:  # 注意要指定 b 参数
    pickle.dump(a, f)
    pickle.dump(b, f)
    pickle.dump(c, f)
    pickle.dump(d, f)
    pickle.dump(e, f)
```

读二进制文件时使用 pickle 模块的 load()函数，它的一般使用形式如下：

```
load (文件对象)
```

其中：文件对象是函数 open()打开的文件对象，从中读取数据对象。函数 load()的返回值是读取的数据对象。

【例 6.6】编写程序，读取并输出当前目录下二进制文件"binary.dat"中的数据。

程序如下：

```
with open("binary.dat", "rb") as f:  # 注意要指定 b 参数
    a = pickle.load(f)
    b = pickle.load(f)
    c = pickle.load(f)
    d = pickle.load(f)
    e = pickle.load(f)
    print(a, b, c, d, e)
```

函数 dump()和 load()会自动处理不同数据对象之间的边界。

【例 6.7】编写程序，首先将三个字符串“程序”“设计”和“基础”写入当前目录下的二进制文件"binary.dat"，然后重新打开该文件读取并输出三个字符串。

程序如下：

```
import  pickle
a = "程序"
b = "设计"
c = "基础"
with open(".\\binary.dat", "wb") as f:
    pickle.dump(a, f)
    pickle.dump(b, f)
    pickle.dump(c, f)
with open(".\\binary.dat", "rb") as f:
    h = pickle.load(f)
    j = pickle.load(f)
    k = pickle.load(f)
    print(h, j, k)
```

6.2.6 文件定位

文件指针是一个用来标示文件当前读/写位置的变量。每当读/写文件时，都从文件指针指向的位置开始读/写。读/写完成后，根据读/写的数据量向后移动文件指针。

文件对象的函数 tell()返回文件指针的当前位置，它的一般使用形式如下：

```
tell()
```

其返回值是文件指针的当前位置。

【例 6.8】当前目录下的文件"log.txt"中保存了一个字符串"python"。编写程序，打开该文件并读取该文件三次，每次读 2 个字符，输出每次读取的内容及读取后的文件指针。

程序如下：

```
with open(".\\log.txt", "r") as f:
    h = f.read(2)
    print(h)
    print("pos = {}".format(f.tell()))
    j = f.read(2)
```

```
        print(j)
        print("pos = {}".format(f.tell()))
        k = f.read(2)
        print(k)
        print("pos = {}".format(f.tell()))
```

文件对象的函数 seek()把文件指针移动到新位置，其一般使用形式如下：

```
    seek(偏移值[,起点])
```

其中，偏移值表示移动的距离；起点表示从哪里开始移动，0 表示从文件头开始，1 表示从当前位置开始，2 表示从文件尾开始，默认值为 0。

【例 6.9】当前目录下的文件"log.txt"中保存了一个字符串"python program"。编写程序，打开该文件并只读取字符串"program"。

程序如下：

```
    with open(".\\log.txt", "r") as f:
        f.seek(7)
        h = f.read(7)
        print(h)
```

6.2.7　读/写 docx 文件和 xlsx 文件

包 docx 是读/写 docx 文件的一个第三方包。这个包的安装命令为

```
    pip  install  python-docx
```

下面介绍使用这个包读/写 docx 文件的几种操作。

1．建立新 Word 文档

建立新文档需要调用 Document 对象的 save 方法，一个 Document 对象代表一个 Word 文档，该方法的参数是保存的文件名。

```
    from docx import Document
    doc = Document()
    doc.save("test.docx")
```

2．添加段落和段落文字

添加段落需要调用 Document 对象的 add_paragraph 方法，该方法的参数是段落文字，返回值是一个 Paragraph 对象。调用 Paragraph 对象的 add_run 方法为该段落添加文字，该方法的参数是添加的文字。add_run 方法的返回值是一个 Run 对象，设置该对象的 bold 属性可以加粗相应的文字，设置该对象的 italic 属性可以把相应的文字设置为斜体。

```
    from docx import Document
    doc = Document()
    p = doc.add_paragraph('无格式的部分')
    p.add_run('加粗部分').bold = True
    p.add_run('无格式')
    p.add_run('斜体部分').italic = True
    doc.save(r".\test.docx")
```

程序执行结束后，文件".\test.docx"中的内容如下所示：

无格式的部分**加粗部分** 无格式 *斜体部分*

3．读取文档的所有段落

Document 对象的 paragraphs 属性是一个包含文档所有 Paragraph 对象的列表对象，一个 Paragraph 对象代表文档的一个段落。对 paragraphs 属性进行循环遍历可以操作文档的所有段落。Paragraph 对象的 text 属性代表该段落的文字。

```
from docx import Document
doc = Document("./test.docx")
for p in doc.paragraphs:
    print(p.text)
```

4．读取文档表格中的文字

Document 对象的 tables 属性是一个包含文档所有 Table 对象的列表对象，一个 Table 对象代表文档的一个表格。Table 对象的 cells 属性是一个包含表格所有_Cell 对象的列表，一个_Cell 对象代表表格的一个单元格。对表格的 cells 属性进行循环遍历可以操作表格的所有单元格。_Cell 对象的 text 属性代表该单元格的文字。

```
from docx import Document
doc = Document("./Python.docx")
for t in doc.tables:
    for c in t._cells:
        print(c.text)
```

包 openpyxl 是读/写 xlsx 文件的一个第三方包，这个包的安装命令如下：

```
pip install openpyxl
```

下面介绍使用该包读/写 xlsx 电子表格的几种操作。

1．创建 Excel 电子表格

建立新文档需要调用 Workbook 对象的 save 方法，一个 Workbook 对象代表一个 Excel 工作簿，该方法的参数是保存的文件名。

```
from openpyxl import Workbook
wb = Workbook()
wb.save("test.xlsx")
```

2．创建工作表

创建工作表需要调用 Workbook 对象的 create_sheet 方法，该方法的参数是工作表的名称。

```
from openpyxl import Workbook
wb = Workbook()
wb.create_sheet("first")
wb.create_sheet("second")
wb.save("test.xlsx")
```

该程序将在工作簿"test.xlsx"中创建两个工作表："first"和"second"。

3．修改单元格的数据

要修改表格数据，需要先调用 load_workbook()函数打开工作表，该函数的参数是 Excel 工作簿的文件名，返回值是一个 Workbook 对象。一个工作簿包含一系列工作表，一个工作表是一个 Worksheet 对象。有三种方法从 Workbook 对象得到其中的一个工作表：第一种是用 Workbook 对象的 get_sheet_by_name 方法，其参数是工作表的名称；第二种是用 Workbook 对象的 worksheets

属性，该属性是一个 Worksheet 对象列表，如 ws = wb.worksheets[1]；第三种是通过索引的方式，下标为工作表的名字，如 ws=wb['first']。

一个 Cell 对象代表一个单元格。得到 Worksheet 对象后，可以通过三种方法获得 Cell 对象并修改其中的数据：第一种是通过索引的方式，下标为单元格的名称，如 ws['A1'] = "数学"；第二种是通过 Worksheet 对象的 cell 方法，该方法的第一个和第二个参数为单元格的行值和列值，均从 1 开始，第三个参数为单元格的值；第三种是通过 Worksheet 对象的 append 方法，其参数为列表，把列表中的值添加到表格中的一行，每个列表元素对应一个单元格。

若提供给单元格的值的形式为"=公式"，则是向单元格中添加一个公式。

```
from openpyxl import Workbook
from openpyxl import load_workbook
wb = load_workbook(r".\test.xlsx")
ws = wb.get_sheet_by_name("first")
ws['A1'] = "数学"
ws['b1'] = "语文"
ws.cell(2, 1, 90)
ws.cell(2, 2, 91)
ws.append([80, 81])
ws['c2'] = "=sum(A2:B2)"
ws['c3'] = "=sum(A3:B3)"
wb.save(r".\test.xlsx")
```

程序执行后，表"first"的内容如下：

数学	语文	
90	91	181
80	81	161

4．读取 Excel 单元格中的数据

获取一个 Cell 对象后，访问 Cell 对象的 value 属性就可读取该单元格中的数据。

```
from openpyxl import Workbook
from openpyxl import load_workbook
wb = load_workbook("./test.xlsx")
ws = wb['first']
print(ws['A1'].value)
print(ws.cell(2, 3).value)
```

6.3　文件与文件夹操作

前两节介绍了文件内容的操作。本节介绍文件与文件夹的操作，如复制、重命名、删除、遍历等。Python 中主要用 os 模块和 os.path 模块来操作文件与文件夹。

6.3.1　os 模块

os 模块是 Python 标准库，它提供了文件与文件夹的相关操作，常用操作如表 6.4 所示。

表 6.4 os 模块的常用文件和文件夹操作

方 法	功 能 说 明
rename(src, dst)	重命名文件或目录，可以实现文件的移动，若目标文件已存在，则抛出异常，不能跨越磁盘或分区
remove(path)	删除指定的文件，要求用户拥有删除文件的权限，并且文件没有只读或其他特殊属性
rmdir(path)	删除文件夹，文件夹有只读属性时也可以删除
mkdir(path)	创建子文件夹
chdir(path)	把 path 设为当前工作目录
getcwd()	返回当前工作目录
listdir(path)	返回 path 目录下的文件和目录列表
startfile(filepath [, operation])	使用关联的应用程序打开指定文件或启动指定应用程序
system()	启动外部程序

```
import  os
os.rename("wt.txt", "wtt.txt")
```

将当前文件夹下的文件"wt.txt"重命名为"wtt.txt"。

```
os.rename("wt.txt", ".\\tmh\\wtt.txt")
```

将当前文件夹下的文件"wt.txt"复制到当前文件夹的子文件夹"tmh"中。

```
os.remove(".\\tmh\\wtt.txt" )
```

删除当前文件夹的子文件夹"tmh"中的文件"wtt.txt"。

```
os.rmdir(".\\tmh")
```

删除当前文件夹的子文件夹"tmh"。

```
os.mkdir("d:\\tmh")
```

在 d 盘根目录下建一个文件夹"tmh"。

```
os.chdir("d:\\tmh")
```

把"d:\\tmh"设置为当前工作目录。

```
os.getcwd()
```

返回当前工作目录。

listdir(path)的功能是返回 path 目录下的文件和目录列表。对该列表进行递归遍历可以遍历文件夹 path 下的所有文件和文件夹。listdir(path)函数返回的列表元素只包含文件夹 path 下的子文件夹和文件的名字，对于其中的元素 p，需要使用 path + "\\" + p 才能得到元素的完整路径名或文件名。

下面的程序段定义了一个函数 traverseDir(path)，参数 path 是一个路径，函数的作用是遍历路径 path 下的所有文件和文件夹，并输出其中的文件和文件夹的完整名称。该函数采用递归的方法实现。

```
from os.path import isfile, isdir
from os import listdir

def traverseDir(path):
    for p in listdir(path):
        subpath = path + "\\" + p
        if isfile(subpath):
            print(subpath)
```

```
            elif isdir(subpath):
                print(subpath)
                traverseDir(subpath)
```

例如，若要遍历文件夹"d:\\python"，则按如下方式调用该函数就可输出它下面的所有文件和文件夹：

```
traverseDir(r"d:\python")
```

6.3.2　os.path 模块

os.path 模块属于 Python 标准库，它提供了关于路径判断、连接及切分的方法。os.path 模块的常用方法如表 6.5 所示。

```
import  os.path
os.path.dirname('D:\\Workspace\\tmh\\t7.txt')
```

返回该路径的文件夹部分'D:\\Workspace\\tmh'。

表 6.5　os.path 模块的常用方法

方　法	功 能 说 明	方　法	功 能 说 明
isdir(path)	判断 path 是否为文件夹	exists(path)	判断文件是否存在
isfile(path)	判断 path 是否为文件	getsize(filename)	返回文件的大小
basename(path)	返回指定路径的最后一个组成部分	join(path, *paths)	连接两个或多个 path
dirname(p)	返回给定路径的文件夹部分	splitext(path)	从路径中分隔文件的扩展名

```
os.path.basename('D:\\Workspace\\tmh\\t7.txt')
```

返回该路径的最后一个组成部分't7.txt'，是该路径的文件名。

```
os.path.basename('D:\\Workspace\\tmh')
```

返回该路径的最后一个组成部分'tmh'，是该路径的最后一个文件夹名。

```
os.path.join('D:\\Workspace', 'tmh')
```

将两个路径连接成一个路径，加上路径分割符，即'D:\\Workspace\\tmh'。

```
os.path.splitext('D:\\Workspace\\tmh\\t7.txt')
```

从路径中分割出文件的扩展名，它的返回值是一个包含两个元素的元组('D:\\Workspace\\tmh\\t7','.txt')。

6.4　编程实例

前面介绍了 Python 编程的基本语法。编程是一个熟能生巧的工作，下面举几个实例，请读者阅读理解后重写一次。编程时一定要自己动手写代码，因为看代码和写代码是两回事。

【例 6.10】BMI（Body Mass Index，体质指数）是国际上常用的量度体重与身高比例的指数，其计算公式为 BMI = 体重（千克）/身高（米）的平方。成人的 BMI 数值判断标准如下所示：过轻，低于 18.5；正常，18.5～23.9；超重，24～27.9；肥胖，大于 28。试编写 BMI 计算程序给出相应的判断，并将测试结果存入 BMI.txt 文件。

```
print('BMI=体重/(身高的平方)')
f = open('BMI.txt', 'a+')                    # a+添加模式打开文件
while  True:                                 # 循环，以便反复测试
```

```
        weight = float(input('体重(千克)='))       # 转浮点型
        if  weight <= 0: break                    # 如输入体重<=0 则结束
        height = float(input('身高(米)='))
        bmi = round(weight/(height**2), 2)        # 计算 BMI，保留 2 位小数
        if   bmi < 18.5:
            s = '过轻'
        elif  bmi <= 23.9:
            s = '正常'
        elif  bmi <= 27.9:
            s = '超重'
        else:
            s = '肥胖'
        info = '体重={} 身高={}  BMI={} 评级:{}\n'.format(weight, height, bmi, s)
        print(info)
        f.write(info)      # 写入文件
    f.close()
```

程序测试时，显示如下（输入体重 75，身高 1.8）。程序可连续输入测试数据，输入体重 0 时结束程序。测试后可打开 BMI.txt 文件查看测试数据。

```
BMI=体重/(身高的平方)
体重(千克)=75
身高(米)=1.8
体重=75.0 身高=1.8  BMI=23.15 评级:正常
体重(千克)=0
```

【例 6.11】每次输入一个数据，连续输入若干数据，输入-1 时结束输入。然后输出数据的最大值、最小值、平均值，按升序输出所有数据。

```
li=[ ]                  # 定义容纳数据的空列表
x=0
while  x != -1:
    x=float(input('请输入一个数据(-1 结束):'))  # 转浮点型
    if  x != -1:
        li.append(x)    # 添加到列表中
    else:
        break           # 如为-1，则跳出循环
if  len(li)==0:         # 判断列表长度
    print('列表为空，没有输入有效数据')
else:
    print('数据个数:', len(li))
    print('最大值:', max(li))
    print('最小值:', min(li))
    print('平均值:', sum(li) / len(li))   # 和/数据个数得到平均值
    print('升序输出：', sorted(li))
```

【例 6.12】猜数游戏。计算机随机产生 1 至 20 之间的一个整数，让用户猜测。计算机根据用户输入的数提示猜测值是偏大还是偏小，最多允许猜 5 次。

```
import  random
num = random.randint(1, 20)      # 产生 1～20 间的随机整数
k = 1                            # 已猜测次数
while  k<=5 :
```

```
        x = int(input('请输入一个[1-20]间的猜测数:'))
        if  num > x:
            print('偏小')
        elif  num < x:
            print('偏大')
        else:
            break               # 如猜中则跳出循环
        k = k+1                 # 次数加 1
    if  k <= 5:
        print('你猜对了。猜测次数:', k)
    else:
        print('你没有猜对。随机数为:', num)
```

【例 6.13】利用字典模拟选票计票统计。

本例没有实际的选票数据，可以使用随机数函数产生模拟的选票。name 是候选人名单，random.choice(name)随机选择一个候选人产生一张选票，共 3000 张。然后遍历所有的选票，将每位候选人的得票情况记录在字典对象中，最后对字典的键值对排序。

```
    import  random
    name = ['甲', '乙', '丙', '丁', '甲']     # 候选人名单，甲写了两次，随机选中概率高
    vote = [ ]                                # 选票列表
    num = 3000                                # 选票总数
    for  x  in  range(num):
        vote.append(random.choice(name))      # 将随机选票加入列表
    di={ }        # 空字典存放统计结果，统计后格式类似{'甲':8, '乙':3}
    for  s  in  vote:
            di[s] = di.get(s, 0) + 1          # 候选人姓名为键，该候选人票数加 1
                  # get(s,0)表示如候选人尚不在字典中，则默认初始票数为 0
    print('总票数: ' , num)
    print('未排序的结果如下:')
    for  x  in  di:
        print(x, di[x])

    #di.items()取出键值对，得到类似于(("甲":8), ("乙":3))的元组数据
    #key=lambda 指定排序的匿名函数，x 代表类似于("甲":8)的数据，则 x[1]对应票数
    li = sorted(di.items(), key=lambda x: x[1], reverse=True)    # 按 x[1]票数逆序排
    print('\n 按票数排序后:')
    for  x  in  li:
        print(x[0], x[1])                     # x[0]姓名，x[1]票数
```

程序执行后的显示结果如下。因为选票是随机产生的，所以每次的测试结果不同。

```
    总票数:  3000
    未排序的结果如下:
    乙 623
    丁 584
    甲 1214
    丙 579

    按票数排序后:
    甲 1214
```

```
乙 623
丁 584
丙 579
```

另一种统计投票的方式是调用 collections 模块中的 Counter 函数，这个函数相当于将上面的字典统计的代码都封装了，用户直接调用就可得到字典统计结果。示例如下：

```
from collections import Counter          # 引入 Counter 函数
counts = Counter(vote)      # 用 Counter 统计 vote 中的数据频次，返回类似的字典数据
print('Counter 函数统计结果：')
for x in counts:
    print(x, counts[x])
print('Counter 函数统计得票前 2 位是：')
for k , v in counts.most_common(2):    # 按键值对返回排名前 2 位的数据
    print(k, v)
```

从 Counter 函数的例子可以发现，调用系统已有的函数会极大地减少编码量，所以编程时首先要看系统有无合适的函数，没有需要的函数时才自定义函数。

6.5 本章小结

本章介绍了 Python 中的文件操作。首先引入了文件的基本概念。接着介绍了文件的基本操作，包括打开文件、关闭文件，文件对象的属性和常用方法，文本文件的读/写方法，二进制文件的读/写方法，以及文件定位的相关操作；着重介绍了读/写 docx 文件和 xlsx 文件的第三方库的使用方法。最后介绍了文件与文件夹操作的相关模块。

习题

1. 以只写方式打开文本文件 file.txt，将一个整数写入该文件，然后关闭文件。用只读方式打开同样的文件，读入一个整数，然后关闭该文件。最后输出读入的整数。
2. 以只写方式打开文本文件 file.txt，将一个整数列表写入该文件，然后关闭文件。用只读方式打开同样的文件，将文件中的所有整数读入一个列表，然后关闭该文件。最后输出读入的整数列表。
3. 编写函数 fun()，将整数 10～30 及它们的平方根写到名为 myfile.txt 的文本文件中，然后按顺序读出并显示在屏幕上。在主程序中调用函数 fun。
4. 编写函数 fun(str, a, d)，将形参给定的三个参数“字符串”“整数”和“实数”按行写入文本文件 myfile.txt，再用字符方式从该文本文件中逐个读出并显示在终端屏幕上。在函数 main 中调用函数 fun。
5. 当前目录下有一个文本文件 test.txt，查找该文件中最长的一行并输出该行的内容。
6. 当前目录下有一个 Word 文档“毕业.docx”，它是一篇毕业论文，读取该文件的内容并输出所有的一级标题。
7. 当前目录下有一个 Excel 工作簿“成绩.xlsx”，它是一个班学生三门课程的成绩。在该工作簿中输入公式，为每名学生计算总分并保存。
8. 编写一个函数，不使用递归调用的方法遍历一个给定路径下的所有文件，并输出各文件的大小。

第 7 章　NumPy 科学计算库

前面介绍了基本的 Python 编程技能，从本章开始介绍 Python 在数据处理、数据分析等方面的应用。Python 生态圈有很多软件包，如 NumPy、Pandas、Matplotlib、Scipy 等，这些软件包相互配合，使 Python 成为目前数据科学领域的首选开发平台。

NumPy（Numerical Python）是一个开源的 Python 科学计算库，是 Python 生态圈中最重要的底层支持库，它支持快速的数组和矩阵运算。NumPy 1.0 发布于 2006 年，目前较新的版本是 1.17，其官网地址为 http://www.numpy.org/。

NumPy 提供的数组乍看之下类似于 Python 的列表，但列表保存的是对象指针，内存消耗大，编程时需要用循环语句处理列表元素，计算速度慢。NumPy 使用内存映射文件来处理数组，以达到最优的数据读写性能。NumPy 的大部分代码是用 C 语言编写的，因此 NumPy 要比纯 Python 代码的速度快得多。NumPy 是在 Python 中进行数据分析、机器学习、人工智能开发的必备工具，是理解学习很多 Python 工具包的基础。

标准 Python 安装包中不含 NumPy，可以执行 `pip install numpy` 命令来安装 NumPy。如果系统安装的是 Anaconda 科学计算平台，那么该平台中默认含有 NumPy。

7.1　NumPy 基础

7.1.1　数组对象特性

一般来说，引入 NumPy 的标准格式为 `import numpy as np`，本章后续都用 np 指代 NumPy。

```
In[1]: import  numpy  as  np     # 引入 NumPy
In[2]: np.__version__            # 显示 NumPy 的版本号，此处前后都是两个下画线
Out[2]: '1.17.0'
In : arr = np.arange(5)          # 生成一个数组，含 5 个元素
In : arr
Out: array([0, 1, 2, 3, 4])
In : type(arr)
Out: numpy.ndarray               # 数组的数据类型
```

NumPy 数组的数据类型是 `numpy.ndarray`，这类数组对象具有很多特性。下面产生一个二维数组对象，并考察这个对象的各种特性。

```
In: b = np.arange(6).reshape(2, 3)   # 生成一个 2x3 的二维数组
In: b
Out:
array([[0,1,2],
       [3,4,5]])
In: b.ndim                           # 数组维度
Out:2
In: b.size                           # 数组中的元素个数
Out:6
In: b.shape                          # 数组形状
```

```
Out: (2,3)
In: b.dtype                          # 数组元素的数据类型
Out:dtype('int32')
In: b.T                              # 转置，将数组的行/列交换
Out:
array([[0, 3],
       [1, 4],
       [2, 5]])
In: b?                               # 查看数组 b 的帮助信息
In: dir(b)                           # 显示数组 b 的属性和方法
In: np.array?                        # 查看 array 帮助信息
```

上面的例子调用np.arange()函数首先创建了一个含有6个元素的一维数组，然后使用reshape()函数将其转换为一个2×3的二维数组。数组对象比较常用的特性如下：

- ndarray.ndim：数组的维度
- ndarray.size：数组元素的总数
- ndarray.shape：数组的形状
- ndarray.dtype：数组中元素的数据类型
- ndarray.T：数组转置

借助这些特性，我们可以快速了解数组的基本情况。

7.1.2 生成数组

NumPy 提供了多种生成数组的方法，可直接用元组、列表来生成数组，也可用内建的各种函数快速生成全0、全1的特殊数组，还可生成各类随机数数组。

1. 使用 array()函数生成数组

可将元组、列表作为参数传递给 array()函数以生成 NumPy 数组，如下面的代码所示：

```
In: b = np.array((1, 2, 3, 4))       # 以元组为参数生成数组
In: b
Out: array([1, 2, 3, 4])
In: b = np.array([10, 20, 30, 40])   # 以列表为参数生成数组
In: b
Out: array([10, 20, 30, 40])
```

列出多个数据时，要注意将数据放入()或[]，即以元组或列表的形式列出多个数据。类似np.array(1,2,3,4)的语句将出现语法错误。

NumPy 的数组对象可调用 tolist()函数转换为 Python 的列表对象。例如，

```
In: lst = b.tolist()            # 数组转换为列表
In: lst
Out: [10, 20, 30, 40]
```

利用 array()函数除了可以生成一维数组，还可以生成二维数组。例如，

```
In: b = np.array([[1, 2] , [3, 4], [5, 6]])  # 生成一个 3x2 的二维数组
In: b
Out:
array([1, 2],
      [3, 4],
      [5, 6] ])
```

```
In: b.shape                 # 形状
Out: (3, 2)
In: b.ndim                  # 维度
Out:2
In: np.sum(b, axis=0)       # 按 0 轴求和
Out: array([9, 12])
In: np.sum(b, axis=1)       # 按 1 轴求和
Out: array([3, 7, 11])
```

二维数组有行和列，在 NumPy 中也称第 0 轴、第 1 轴，在很多函数中以参数 axis=0 或 axis=1 的形式指明。

2. 使用 arange()函数生成数组

arange()是 NumPy 中的函数，它类似于 Python 中的 range()，调用格式为 arange(start, stop, step)，功能是生成指定范围内的等差数组。range()函数的步长参数 step 必须为整数，arange()函数的步长参数 step 可以为小数。例如，

```
In: b = np.arange(1, 5)          # 生成 1 至 4（不含终值 5）的数组，步长默认为 1
In: b
Out:array([1, 2, 3, 4])
In: np.arange(1, 9, 2)           # 步长为 2，产生含 1,3,5,7（不含终值 9）的等差数组
Out: array([ 1, 3, 5, 7])
In: np.arange(1, 4, 0.5)         # 步长为 0.5
Out: array([1. , 1.5, 2. , 2.5, 3. , 3.5])

In: np.arange(4000).reshape(4, 1000) # 生成 4x1000 的大数组，默认只显示周边元素
Out:
array([[   0,    1,    2, ...,  997,  998,  999],
       [1000, 1001, 1002, ..., 1997, 1998, 1999],
       [2000, 2001, 2002, ..., 2997, 2998, 2999],
       [3000, 3001, 3002, ..., 3997, 3998, 3999]])
```

如果数组比较大而不方便显示输出数组的全部元素，那么 NumPy 会自动跳过数组中间的部分而只显示数组四周的元素。

已有的数组还可使用 repeat()函数或 np.tile()函数进行重复，以构建更大的数组。例如，

```
In: b = np.arange(3)
In: b
Out: array([0, 1, 2])
In: b.repeat(4)                  # 将元素重复 4 次
Out: array([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2])

In: b = np.arange(6).reshape(2, 3)
In: b
Out:
array([[0, 1, 2],
       [3, 4, 5]])
In: np.tile(b, 3)                # 将数组重复 3 次
Out:
array([[0, 1, 2, 0, 1, 2, 0, 1, 2],
       [3, 4, 5, 3, 4, 5, 3, 4, 5]])
```

3. 使用 ones()、zeros()等函数生成数组

利用 ones()、zeros()、ones_like()、zeros_like()等函数可以生成全 0 和全 1 的数组。例如，

```
In: np.ones(5)              # 生成含有 5 个元素的一维全 1 数组
Out: array([1., 1., 1., 1., 1.])
In: np.ones((2, 3))         # 生成 2x3 的二维全 1 数组
Out:
array([[1., 1., 1.],
       [1., 1., 1.]])

In: np.zeros((2, 3))        # 生成 2x3 的二维全 0 数组
Out:
array([[0., 0., 0.],
       [0., 0., 0.]])

In: np.full(5, 1.2)         # 生成 5 个元素都为 1.2 的一维数组
Out: array([1.2, 1.2, 1.2, 1.2, 1.2])

In: b = np.arange(24).reshape(4, 6)  # 生成 4x6 的二维数组 b
In: c = np.ones_like(b)              # 生成维数和 b 相同的全 1 数组
In: c = np.zeros_like(b)             # 生成维数和 b 相同的全 0 数组

In: np.eye(3)                        # 生成单位矩阵数组
Out:
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])

In: np.diag([1, 3, 5])               # 生成对角线矩阵数组
Out:
array([[1, 0, 0],
       [0, 3, 0],
       [0, 0, 5]])
```

4. 使用 linspace()函数生成等差数组

使用 linspace(start, stop, num)函数可以创建一个等差数组，调用时需要指定等差数组的初值、终值和数据个数，但不需要指定差值。例如，

```
In: np.linspace(1, 2, 5)             # 在区间[1, 2]内等间距生成 5 个数据，含终值 2
Out: array([1.  , 1.25, 1.5 , 1.75, 2.  ])

In: np.linspace(-np.pi, np.pi, 10)   # 在区间[-π, +π]内等间距生成 10 个数
Out:
array([-3.14159265, -2.44346095, -1.74532925, -1.04719755, -0.34906585,
        0.34906585,  1.04719755,  1.74532925,  2.44346095,  3.14159265])
In: np.linspace(1, 2, 5, endpoint=False)   # 不含终值
Out: array([1. , 1.2, 1.4, 1.6, 1.8])
```

linspace()生成数组时默认包含终值，若设置 endpoint=False，则不包含终值。

5．使用随机函数生成数组

NumPy 提供了众多的随机函数用于快速便捷地生成各类随机数数组。例如，

```
In: np.random.rand(3, 4)    # 3x4 数组，rand()生成在区间[0, 1)内均匀分布的小数
Out:
array([[0.38457501,  0.51763253,  0.07178795,  0.20844094],
       [0.51057372,  0.07895232,  0.92448913,  0.70173132],
       [0.05224563,  0.95209293,  0.96886966,  0.54983132]])
```

上面的数据显示了很多位小数，NumPy 可以自行设置小数的显示精度。例如，

```
In: np.set_printoptions(precision=3)      # 设置小数部分显示 3 位
In: np.set_printoptions?                  # 显示该命令的各种可设置选项

In: np.random.randint(1, 100, 5)          # 生成区间[1,100)内的 5 个随机整数
Out: array([70, 70, 89, 76, 66])
In: np.random.randn(10, 20)               # 生成符合标准正态分布 N(0,1)的 10x20 数组
# 下例参数中的 loc 是均值，scale 是标准差
In: np.random.normal(loc=1, scale=2, size=(3,4)) # 生成符合正态分布 N(1,4)的 3x4 数组
```

7.1.3　NumPy 的数据类型

Python 支持的数据类型有整型、浮点型和复数型，这些类型难以满足科学计算的实际需求，因此 NumPy 增加了很多与科学计算有关的数据类型，常用的数据类型如表 7.1 所示。创建数组时以参数形式 dtype='类型'指定特定的数据类型。

表 7.1　NumPy 的常用数据类型

类　型　名	描　　述
bool	布尔类型（值为 True 或 False）
int	由所在平台决定其精度的整数（一般为 int32 或 int64）
int8	8 位整数，范围为-128～127
int16	16 位整数，范围为-32768～32767
int32	32 位整数，范围为-2^{31}～$2^{31}-1$
int64	64 位整数，范围为-2^{63}～$2^{63}-1$
uint8	无符号 8 位整数，范围为 0～255
uint16	无符号 16 位整数，范围为 0～65535
float16	半精度浮点数（16 位）：1 位符号位，5 位指数，10 位尾数
float32	单精度浮点数（32 位）：1 位符号位，8 位指数，23 位尾数
float64	双精度浮点数（64 位）：1 位符号位，11 位指数，52 位尾数
complex64	复数，分别用两个 32 位浮点数表示实部和虚部

创建数组时，不同的创建函数具有不同的默认数据类型，可以在创建时用 dtype 参数指定类型。例如，

```
In: b = np.arange(5)    # arange 函数默认的数据类型是 int32
In: b.dtype
Out: dtype('int32')
In: b[0] = 8.5          # 由于是 int32 类型，所以在赋小数值时只保留整数 8
In: b
Out: array([8, 1, 2, 3, 4])      # 注意 b[0]是整数 8
```

```
In: c = np.arange(5, dtype='float32')     # 指定数据类型为 float32
In: c[0] = 8.5                            # 因为是 float32 类型，所以可正确保存小数
In: c
Out: array([8.5, 1. , 2. , 3. , 4. ], dtype=float32)

In: d = np.ones(5)                        # ones 函数默认类型为 float64
In: d.dtype
Out: dtype('float64')
In: np.full(5, 1, dtype='bool')           # 创建布尔数组，1 为真，全为 True
Out: array([ True,  True,  True,  True,  True])
In: np.full(5, 0, dtype='bool')           # 0 为假，全为 False
Out: array([False, False, False, False, False])
```

需要注意的是，ndarray 要求每个数组元素的数据类型是一样的，而 Python 列表允许包含不同数据类型的元素。数据类型之间可用 astype()函数进行转换。转换时原数组不变，生成一个新数组。例如，

```
# 将数组 c 的数据由 float32 转换为 int32 类型，存为数组 b，c 保持不变
In: b = c.astype('int32')
# 要直接修改原数组，可将转换结果赋值给原数组
In: c = c.astype('int32')
In: d = np.array([1, 2, 0, 0, 3])
In: d.astype('bool')        # 转换为 bool 数组。非 0 为 True，0 为 False
Out: array([True, True, False, False, True])
```

7.2 存取数组元素

7.2.1 基本索引和切片操作

NumPy 支持一维和多维数组。存取一维数组的方法类似于访问 Python 列表，可单个索引或切片访问。例如，

```
In: b = np.arange(5, 10)
In: b
Out:array([5, 6, 7, 8, 9])
In: b[1]               # 单个索引下标访问元素
Out:6
In: b[-1]              # 访问末尾元素
Out:9
In: b[4] = 20          # 将第 4 个元素修改为 20
In: b
Out:array([5, 6, 7, 8, 20])

In: b[[0, 1, 3]]       # 访问第 0,1,3 个元素，注意多个下标要放在[]内
Out: array([5, 6, 8])

In: c = b[0:3]         # 切片访问，访问第 0,1,2（不含第 3）个元素
In: c
Out:array([5,6,7])
```

NumPy 的切片语法和列表的切片语法相同，但要注意 NumPy 的数组切片和 Python 的列表切片在内存管理机制上是不同的。如果列表中的元素是普通数据而非列表，那么一般的列表切片是复制元素，切片和原列表在内存中是独立的，两者互不影响。NumPy 的数组切片产生的是一个视图，视图和原数组指向同一个内存空间，两者相互影响。NumPy 这样处理的目的是为了提升数组的操作性能。可以设想，对于一个庞大的数组，如果 NumPy 在切片时采用复制的方式，那么将极大地耗费内存空间。

```
# 下例演示 NumPy 数组切片和原数组共享内存
In: c = b[0:3]
In: c[0] = 100                          # 将 c[0]修改为 100
In: c
Out:array([100,6,7])
In: b                                   # 再查看原数组 b
Out: array([100, 6, 7, 8, 20])          # 可见 b[0]也被修改为 100
```

上面的代码中，数组 c 是数组 b 的切片，两者共享内存。将 c 的第 0 个元素修改为 100 时，原数组 b 也被修改。如果希望避免这种影响，可用 c=b[0:3].copy()这样的复制语句，复制得到的数组 c 是和原数组 b 分离的新数组。NumPy 的很多操作一定要注意区分是视图还是复制。

7.2.2　二维数组的索引操作

二维数组分成行和列两个维度，其索引下标要比一维数组更灵活，可以使用[行,列]、[行]、[:,列]等下标形式。

```
In: b = np.arange(12).reshape(3, 4)  # 生成一个 3x4 的二维数组
In: b
Out:
array([[0, 1,  2,  3],
       [4, 5,  6,  7],
       [8, 9, 10, 11]])
In: b[1, 2]                 # 访问数组第 1 行的第 2 个元素，即 6
Out: 6
In: b[1][2]                 # 同上，坐标也可写为[1][2]
Out: 6
In: b[1, 1:3]               # 第 1 行的第 1～2 个元素
array([5, 6])

In: b[1]                    # 只给出行坐标，省略列坐标，访问第 1 行的所有元素
Out: array([4, 5, 6, 7])
In: b[:, 1]                 # 行坐标为:代表所有的行，访问第 1 列的所有元素
Out: array([1, 5, 9])
In: b[1:3, 1:4]             # 取第 1～2 行:第 1～3 列的数据块
Out:
array([[5,  6,  7],
       [9, 10, 11]])
```

假设要选取数组 b 中的 3、6、9 数据项，这三个数据的索引坐标为[0,3]、[1,2]、[2,1]，那么访问这三个数的语句可写为 b[[0,1,2],[3,2,1]]，即把行、列坐标分别放在列表中排列，这种格式也称花式索引。花式索引得到的是复制数组，而不是数组视图。

```
In: b[[0, 1, 2], [3, 2, 1]]     # 花式索引
Out: array([3, 6, 9])
```

```
In: b[[2, 0]]                    # 取第 2 行、第 0 行两行的数据，注意 2,0 在[]内
Out:
array([[8, 9, 10, 11],
       [0, 1,  2, 3]])
In: b[:, [1, 3]]                 # 取第 1 列、第 3 列元素
Out:
array([[1,  3],
       [5,  7],
       [9, 11]])
```

7.2.3 布尔索引

布尔索引用一个布尔数组来索引数组元素，选取 True 值对应位置的数据。

```
# 设置随机数种子 7，保证每次测试时按序生成相同的随机数，测试结果可重现
In: np.random.seed(7)
In: b = np.random.randint(40, 100, size=10)    # 生成 10 个区间[40,100)内的随机整数
In: b
Out: array([87, 44, 65, 94, 43, 59, 63, 79, 68, 97])
In: c = b < 60                                 # 生成一个布尔数组
In: c
Out:
array([False, True, False, False, True, True, False, False, False, False])
In: b[c]                                       # 利用布尔索引取数据，得到<60 的数据
Out: array([44, 43, 59])
```

上例中的 c=b<60 将生成一个布尔数组 c，其中第 1、4、5 个值为 True，执行 b[c]时将选取布尔数组中 True 值对应位置的元素。采用布尔索引访问数组元素时，要保证布尔数组和数据数组的维度相同。

利用布尔索引选取数据时，还可结合 &（与）、|（或）、~（非）等逻辑运算进行更复杂的数据选取。

```
# &(与)，显示区间[60, 80]内的数据。注意此处要用&而不能用 and
In: b[(b >= 60) & (b <= 80)]
Out: array([65, 63, 79, 68])
In: b[(b < 60) | (b > 90)]           # |或
Out: array([44, 94, 43, 59, 97])
In: b[~(b < 60)]                     # ~非，显示>=60 的数据
Out: array([87, 65, 94, 63, 79, 68, 97])
```

NumPy 还提供一个按条件选取数据的 where(condition, x, y)函数，其中的参数 condition 是一个布尔数组，函数将按其逻辑值分别选择数组 x，y 中的值，示例如下。

```
In: x = np.array([0, 1, 2, 3, 4])
In: y = np.array([10, 11, 12, 13, 14])
In: c = np.array([True, True, False, False, True])
In: np.where(c, x, y)       # 从 x 中取第 0,1,4 个值，从 y 中取第 2,3 个值
Out: array([0, 1, 12, 13, 4])
```

where()函数后面的两个参数可以不是数组而是单个值。

```
In: np.where(x>2, 1, 0)     # 数组 x 中的数>2 时取 1，否则取 0
Out:array([0, 0, 0, 1, 1])
```

7.3　数组运算和排序

7.3.1　数组和单个数据的运算

数组和单个数据的运算规则很简单，即对单个数据和数组的每个元素进行运算。Python 的列表一般需要编写循环代码才能对整个列表元素进行计算，而数组不需要循环语句就可实现同样的操作。

```
In: b = np.arange(5)
In: b
Out:array([0, 1, 2, 3, 4])
In: b + 2              # 将每个数组元素和单个数据相加
Out: array([2, 3, 4, 5, 6])
In: b * 3              # 将每个数组元素和单个数据相乘
Out: array([0, 3, 6, 9, 12])
In: b / 2              # 将每个数组元素和单个数据相除
Out: array([0. , 0.5, 1. , 1.5, 2. ])
```

7.3.2　数组和数组的运算

1．数组形状相同时的运算

当两个数组的形状相同时，各位置上的元素对位计算。

```
In: a1 = np.arange(5)
In: a1
Out: array([0, 1, 2, 3, 4])
In: a2 = np.arange(5, 10)        # a1, a2 数组形状相同
In: a2
Out: array([5, 6, 7, 8, 9])
In: a1 + a2                      # 数组+数组，各个元素对位相加
Out: array([5,  7,  9, 11, 13])

In: a1 * a2                      # 数组*数组
Out: array([0,  6, 14, 24, 36])
In: a1 / a2                      # 数组/数组
Out: array([0.        , 0.16666667, 0.28571429, 0.375     , 0.44444444])
```

2．数组形状不同时的广播运算

当两个数组的形状不同时，NumPy 将按照广播规则进行运算。

```
In: a1 = np.arange(4)                    # a1 是一维数组
In: a1
Out: array([0, 1, 2, 3])
In: a2 = np.arange(12).reshape(3, 4)     # a2 是 3x4 的二维数组
In: a2
Out:
array([[0,  1,  2,  3],
       [4,  5,  6,  7],
       [8,  9, 10,  11]])
```

```
In: a1.shape,  a2.shape
Out: ((4,), (3, 4))                          # a1, a2 数组形状不同
In: a1 + a2                                  # a1+a2 将使用广播规则运算
Out:
array([[0,  2,  4,  6],
       [4,  6,  8, 10],
       [8, 10, 12, 14]])
```

上例中的 a1 是一维数组，a2 是 3x4 的二维数组，两者的维数不同。a1 的维数较小，NumPy 在 a1 的维数前面加 1 补齐，可近似认为 a1 的维数变为(1,4)。现在 a1 的第 0 维是 1，a2 的第 0 维是 3，NumPy 将把 a1 的第 0 维扩充为 3，相当于将 a1 的维数变为(3,4)，可认为 a1 在内存中被调整为如下形状：

```
array([[0,  1,  2, 3],
       [0,  1,  2, 3],
       [0,  1,  2, 3]]
```

这样，a1 和 a2 的形状一致，就可以完成 a1+a2 的运算。

广播规则具体如下：

（1）参加运算的数组都向维数最大的数组看齐，维数较小的数组在前面加 1 补齐。

（2）结果数组的形状（shape）取各运算数组的各个轴上的最大值。

（3）若运算数组的某个轴的长度为 1，则该轴可扩充为结果数组的对应轴的长度。若轴的长度不为 1，则不能扩充。

（4）检查扩充后所有运算数组对应的轴的长度。若长度都相同则符合规则可以计算，否则违反规则无法计算。

下面再来考察一个(3,1)和(1,2)数组的广播运算示例。

```
In: b1 = np.arange(3).reshape(3, 1)
In: b2 = np.arange(10, 12).reshape(1, 2)
In: b1
Out:
array([[0],
       [1],
       [2]])
In: b2
Out: array([[10, 11]])

In: b1 + b2
Out:
array([[10, 11],
       [11, 12],
       [12, 13]])
```

b1 的形状为(3,1)，b2 的形状为(1,2)，两者都是二维数组，运算时无须再增加各自的维数。按照上面的第 2 条规则，每个轴取最大值，因此结果数组的形状应为(3,2)。按第 3 条规则，可近似认为 b1 沿水平方向扩充为

```
array([[0, 0 ],
       [1, 1 ],
       [2, 2 ])
```

b2 沿竖直方向扩充为

```
        array([[10, 11 ],
               [10, 11 ],
               [10, 11 ])
```

经过广播规则处理后，b1 和 b2 两个数组的形状相同，可以执行数组与数组的运算。

不满足广播规则的数组将无法计算。例如，2×3 和 3×2 的两个数组将无法计算，2×3 和 4×3 的两个数组也无法广播计算。

7.3.3　数组排序

NumPy 数组自带 sort()排序方法，它可把数据从小到大排列。

```
In: np.random.seed(7)  # 设置随机数种子 7，保证每次测试时按序生成相同的随机序列
In: b = np.random.randint(1, 20, size=10)  # 生成区间[1,20)内的 10 个随机整数
In: b
Out: array([16,  5,  4,  8, 15,  9, 15, 11,  9,  8])
In: c = b.copy()              # 将数组 b 复制一份，用于后续的测试
In: b.sort()                  # 排序后直接改变了 b 的数据顺序
In: b
Out: array([ 4,  5,  8,  8,  9,  9, 11, 15, 15, 16])
```

如果排序时不想改变数组自身，那么可以使用 np.sort()函数。

```
In: b = c.copy()              # 从备份数组 c 中恢复原数组
In: np.sort(b)                # np.sort 排序后返回一个新数组，数组 b 自身不变
Out: array([ 4,  5,  8,  8,  9,  9, 11, 15, 15, 16])
```

还有一个常用的排序函数 np.argsort()，它可以返回一个代表原数据顺序的有序下标数组。

```
In: b
Out: array([16,  5,  4,  8, 15,  9, 15, 11,  9,  8])
In: np.argsort(b)              # 排序后返回代表原数据顺序的有序下标
Out: array([2, 1, 3, 9, 5, 8, 7, 4, 6, 0], dtype = int64)
```

argsort()的排序结果表明数组 b 中最小的数是第 2 个，其次是第 1 个，最大的数是第 0 个。

```
In: arg = np.argsort(b)      # 得到排序的有序下标数组
In: c = b[arg]               # 利用返回的下标数组抽取得到有序数组
In: c
Out: array([4,  5,  8,  8,  9,  9, 11, 15, 15, 16])
In: b[arg[0]]                # 抽取最小的数
Out: 4
In: b[arg[-1]]               # 抽取最大的数
Out: 16
```

NumPy 不提供类似列表逆序排序时所用的 reverse=True 参数。要将数组逆序排列，可使用一个如下所示的小技巧：

```
        In: b[np.argsort(-b)]       # 注意此处是-b
        Out: array([16, 15, 15, 11,  9,  9,  8,  8,  5,  4])
```

其中-b 是对数组 b 取反，这样 np.argsort(-b)将返回一个代表原数组值从大到小排列的索引下标，再以此索引下标对原数组抽取，就得到了逆序排列。

在对二维数组排序时务必要注意指定排序的轴。二维数组有行和列两个轴，分别用参数 axis=0 和 axis=1 指定。

```
        In: np.random.seed(7)
        In: b = np.random.randint(1, 50, size=15).reshape(3, 5)
```

```
In: b
Out:
array([[48,  5, 26,  4, 20],
       [24, 40, 29, 15, 24],
       [ 9, 26, 47, 43, 27]])

In: np.sort(b)              # 默认按最后一个轴（axis=1）排列，在水平方向上排序
Out:
array([[ 4,  5, 20, 26, 48],
       [15, 24, 24, 29, 40],
       [ 9, 26, 27, 43, 47]])

In: np.sort(b, axis=0)      # 按 axis=0 排列，在竖直方向上排序
Out:
array([[ 9,  5, 26,  4, 20],
       [24, 26, 29, 15, 24],
       [48, 40, 47, 43, 27]])
```

7.4 NumPy 的函数

NumPy 提供了很多计算函数，这些函数也称 ufunc 通用函数。通用函数可一次性地对整个数组进行计算，而无须编写循环语句处理。

7.4.1 常用函数

NumPy 的常用函数有 max()、min()、ptp()、sum()、var()、std()等（见表 7.2），它们可方便快速地对数组进行统计计算。

表 7.2 NumPy 的常用函数

函 数 名	功 能
max, min	返回数组的最大值、最小值
ptp	返回数组的极差，即最大值 - 最小值
sum, mean	返回和、平均值
var, std, median	返回方差、标准差、中位数
exp, exp2	返回数组以 e 或 2 为底的指数运算结果
log, log2, log10	返回自然对数、以 2 为底的对数、常用对数
sqrt	返回平方根
sin, cos, tan	三角函数
arcsin, arccos, arctan	反三角函数
floor	向下取整
ceil	向上取整
trunc	返回整数，截掉小数部分
prod	将数组元素累乘
cumprod	累乘并返回中间结果
unique	返回不重复的数组元素

（续表）

函 数 名	功　能
nansum、nanmean	剔除 nan 值后求和、求平均值
isnan	检查每个数据值是否为 nan，返回布尔数组
np.maximum(arr1, arr2)	元素级最大值，数据两两比较返回较大值
np.minimum(arr1, arr2)	元素级最小值，数据两两比较返回较小值
np.quantile(arr,p)	返回分位点 p 对应的分位数

```
In: np.random.seed(7)
In: b = np.random.randint(1, 20, size=8)
In: b
Out: array([16,  5,  4,  8, 15,  9, 15, 11])
In: np.max(b), np.min(b)                    # 最大值、最小值
Out: (16, 4)
In: np.ptp(b)                               # ptp 返回数据极差，即最大值-最小值
Out: 12
In: np.quantile(b, [0.25, 0.5, 0.75])       # 返回分位点 25%, 50%, 75%对应的分位数
Out: array([ 7.25,  10. ,  15. ])
In: np.sum(b), np.mean(b)                   # 求和、平均值
Out: (83, 10.375)
```

上述函数调用时可写为“np.函数名(数组)”的形式。对于很多常用的函数，NumPy 也将其包装为数组对象自身的方法，所以也可写为“数组.函数名()”的形式，示例如下。

```
In: b.max(),  b.min(),  b.ptp(),  b.sum(),  b.mean()
Out: (16, 4,  12,  83,  10.375)
In: b.argmax(), b.argmin()            # 返回最大值，最小值所对应的索引下标
Out: (0, 2)
In: b.var(), b.std(), np.median(b)    # 方差，标准差，中位数
Out: (18.984375,  4.357106264483344 , 10.0)
In: np.var(b, ddof=1)                 # 无偏方差
Out: 21.696428571428573
```

NumPy 默认计算的是有偏方差。要计算无偏方差，需增加参数 ddof = 1。

```
In: np.set_printoptions(precision=3)      # 设置小数显示 3 位
In: np.sin(b)                             # 三角函数
Out: array([-0.288, -0.959, -0.757,  0.989,  0.65 ,  0.412,  0.65 , -1.   ])

In: np.sqrt(b)                            # 开平方
Out: array([4.   , 2.236, 2.   , 2.828, 3.873, 3.   , 3.873, 3.317])
```

对于二维数组，函数调用时还可指定计算的轴 axis，示例如下。

```
In: b = np.arange(10).reshape(2, 5)
In: b
Out:
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
In: np.sum(b)               # 未指定轴，计算所有数据的和
Out: 45
In: np.sum(b, axis=0)       # 指定轴 axis=0，沿竖直方向求和
Out: array([ 5,  7,  9, 11, 13])
```

```
In: np.sum(b, axis=1)       # 指定轴 axis=1，沿水平方向求和
Out: array([10, 35])
```

有时我们得到的数据可能有缺失值。NumPy 用 nan(Not a Number)表示缺失值或空值。如果数组中含有 nan 值，那么一般函数的运算结果就是 nan，如下所示。

```
In: b = np.array([1, 2, np.nan, np.nan, 3])    # 构造含有 nan 值的数组
In: np.sum(b)                                  # 数组含有 nan 值时运算结果为 nan
Out: nan
```

要在运算中剔除 nan 值的影响，可采用类似于 nansum()、nanmean()的函数。

```
In: np.nansum(b),  np.nanmean(b)               # 排除 nan 值求和、求平均值
Out: (6.0, 2.0)

In: type(None),  type(np.nan)                  # 比较 None 和 np.nan 的数据类型
Out: (NoneType, float)
```

np.nan 和 Python 自带的 None 是不同的，测试数据类型可知，None 是 NoneType 型的，而 np.nan 是 float 型的，它代表空数据。数据分析中对空数据的处理非常重要，NumPy 对 nan 的支持较少，下一章将要介绍的 Pandas 库对 nan 数据提供了更全面的支持。

NumPy 提供了 isnan()函数，用于检查数组中各元素的值是否为 nan。

```
In: np.isnan(b)
Out: array([False, False, True, True, False])
```

结果显示数组 b 的第 2、3 个元素为 nan 值。

```
In: np.isnan(b).sum()  # 计算 nan 值的个数。True 被视为 1
Out: 2
```

下面介绍 NumPy 的累乘函数 prod()、cumprod()和累加函数 cumsum()。

```
In: c = np.arange(1, 6)
In: c
Out: array([1, 2, 3, 4, 5])
In: np.prod(c)              # 将数组 c 中的元素累乘，即计算 1*2*3*4*5
Out: 120
In: np.cumprod(c)           # 累乘，同时给出累乘的中间结果
Out: array([  1,   2,   6,  24, 120], dtype = int32)
In: np.cumsum(c)            # 累加，同时给出累加的中间结果
Out: array([ 1,  3,  6, 10, 15], dtype = int32)
```

np.unique()用于返回不含重复值的数组。

```
In: b = np.array([1, 3, 3, 5, 5, 7, 8])   # 数组 b 中有重复元素
# np.unique 每个元素只取一个，返回不含重复元素的新数组
In: np.unique(b)
Out: array([1, 3, 5, 7, 8])
```

NumPy 的非 0 测试函数 np.all()和 np.any()。

```
In: np.all(np.array([1, 3, 5]))      # 测试数组中的元素是否都为非 0 值，即真值
Out: True                            # 元素都为非 0 值，返回 True
In: np.all(np.array([1, 3, 0]))
Out: False                           # 数组含有 0，返回 False
In: np.any(np.array([0, 0, 2]))      # 测试数组中是否包含非 0 值
Out: True                            # 含非 0 元素，返回 True
In: np.any(np.array([0, 0, 0]))
Out: False                           # 不含非 0 元素，返回 False
```

NumPy 还提供实现矩阵乘法运算的 dot()函数，如下所示。

```
In: a = np.arange(6).reshape(2, 3)
In: a
Out:
array([[0, 1, 2],
       [3, 4, 5]])
In: b = np.arange(6, 12).reshape(3, 2)
In: b
Out:
array([[ 6,  7],
       [ 8,  9],
       [10, 11]])
In: np.dot(a, b)        # 2x3 和 3x2 数组相乘，结果为 2x2 数组
Out:array([[ 28,  31],
           [100, 112]])
```

7.4.2　随机函数

NumPy 提供很多随机函数，这些函数主要用于数据测试和统计研究，如表 7.3 所示。随机函数都位于 NumPy 的 random 模块中，调用语法为“np.random.随机函数(参数)”。Python 中的随机数都是按照一定的算法生成的伪随机数。如果每次都先设置相同的随机数种子值（seed），那么就能保证随机数按顺序生成的不变性。

表 7.3　NumPy 的常用随机函数

函　数　名	函数描述
seed	设置随机数种子
rand(d0, d1, …, dn)	返回[0,1)间的符合均匀分布的随机小数数组
randn(d0, d1, …, dn)	返回符合标准正态分布 N(0, 1)的随机小数数组
randint(low, high, size)	返回 [low, high) 间的随机整数数组
random(size = (m, n))	返回[0,1) 间的 m x n 维度的随机小数数组
choice(a, n)	从数组 a 中随机选择 n 个数
normal(loc, scale, size)	返回符合正态分布 N(loc, $scale^2$)的随机数
uniform(low, high, size)	产生指定区间均匀分布的样本值

下面的例子演示了设置相同的 seed 后随机数生成的不变性：

```
In: np.random.seed(7)
In: np.random.rand(2)       # 返回区间[0,1)内的 2 个随机小数
Out: array([0.07630829, 0.77991879])

In: np.random.rand(2, 3)    # 返回 2x3 二维随机小数数组
Out:
array([[0.43840923, 0.72346518, 0.97798951],
       [0.53849587, 0.50112046, 0.07205113]])
In: np.random.random((2, 3))          # 返回 2x3 二维随机小数数组
In: np.random.randn(20)               # 返回 20 个标准正态分布 N(0,1)随机数
In: np.random.randint(1, 100, 5)      # 返回区间[1,100)内的 5 个随机整数
```

```
In: b=np.arange(1, 20, 2)
In: b
Out: array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19])
# 从 b 中随机抽取 5 个数
In: np.random.choice(b, 5, replace=False)      # False 不允许重复，类似无放回抽样
Out: array([ 9,  5, 13,  3,  7])
In: np.random.choice(b, 5, replace=True)       # True 允许重复，类似有放回抽样
Out: array([17, 19, 13,  9, 19])

In: np.random.uniform(low=2, high=3, size=10)  # 生成区间[2,3)内的 10 个随机小数
In: np.random.uniform(1, 4, size=(2,3))        # 生成 2x3 二维数组，值在区间[1,4)内
Out:
array([[2.39963287, 1.81662985, 2.75073184],
       [3.17581547, 2.54634477, 3.461999  ]])
In: np.random.normal(loc=3, scale=4, size=100) # 符合正态分布 N(3,16)的随机数
```

7.4.3 集合函数

研究两个数组中的数据是否存在某种集合关系时，可以使用 NumPy 提供的集合函数。

```
In: x = np.arange(5)
In: x
Out: array([0, 1, 2, 3, 4])
In: y = np.arange(3, 8)
In: y
Out: array([3, 4, 5, 6, 7])

In: np.intersect1d(x, y)    # 交集
Out: array([3, 4])
In: np.union1d(x, y)        # 并集
Out: array([0, 1, 2, 3, 4, 5, 6, 7])
In: np.in1d(x, y)           # 得到一个表示 x 的元素是否包含于 y 的布尔型数组
Out: array([False, False, False,  True,  True])

In: np.setdiff1d(x, y)      # 集合的差（即元素在 x 中但不在 y 中）
Out: array([0, 1, 2])

In: np.setxor1d(x, y)       # 集合对称差（存在于一个数组中但不同时存在于两个数组中）
Out: array([0, 1, 2, 5, 6, 7])
```

7.4.4 多项式

NumPy 可以构造多项式，这些多项式可代入数值进行计算，还可求微分、积分、解方程。例如，一元二次多项式 $y = x^2 - 3x - 4$ 可构造如下。

```
In: y = np.poly1d([1, -3, -4])  # 以列表形式给出多项式的系数，y 是构造得到的多项式
In: y
Out: poly1d([ 1, -3, -4])
In: y([0.5, 1, 1.5])            # 分别计算 x 取 0.5, 1, 1.5 时多项式的值
Out: array([-5.25, -6.  , -6.25])
```

```
In: np.roots(y)                  # 计算方程 y = x^2 - 3x - 4 的根
Out: array([ 4., -1.])           # 根为 4 和-1

In: p = np.poly1d([2, 4, 0, 3])  # 构造 p = 2x^3 + 4x^2 + 3
In: z = p+y                      # p, y 两个多项式相加
In: z
Out: poly1d([ 2,  5, -3, -1])    # 表示 z = 2x^3 + 5x^2 - 3x - 1
In: z.deriv()                    # 微分
Out:poly1d([6, 10, -3])
In: z.integ()                    # 积分
Out: poly1d([ 0.5    ,  1.66666667, -1.5     , -1.    ,  0.   ])
```

除了基本的多项式运算，NumPy 还提供 polyfit()函数进行多项式拟合。拟合时需要提供自变量和因变量两组值，并指定拟合多项式的最高次方。示例如下。

```
In: x = np.arange(1, 21)
In: y1 = 3 * x + 2 + np.random.rand(20)        # 构造测试数据 y1 并加入随机干扰值
In: np.polyfit(x, y1, 1)                       # 按一次多项式拟合，拟合系数如下
Out: array([2.99028312, 2.5900484 ])

In: y2 = x**2 + 3 * x + 1 + np.random.rand(20) # 构造测试数据 y2
In: np.polyfit(x, y2, 2)                       # 按二次多项式拟合，拟合系数如下
Out: array([1.00000149, 3.01755285, 1.32390592])
```

7.5　数组组合和文件存取

7.5.1　改变数组的维度

NumPy 提供多种方法来调整数组的维度，最常用的是 reshape()方法。

```
In: b = np.arange(12)        # b 是一维数组，共 12 个数据
In: c = b.reshape(3, 4)      # 调整为 3x4 的二维数组并赋给 c，b 保持不变
In: c
Out:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

#下例中的-1 表示该轴的值由 NumPy 自动计算。第 1 轴的值为 6，所以第 0 轴的值为 12/6=2
In: d = c.reshape(-1, 6)
In: d.shape
Out:(2,6)
In: d
Out:
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
```

注意 reshape()得到的数组是原数组的视图，两者共享内存。除了使用 reshape()方法，还可以使用如下两种方法直接设置数组的维度。修改后数组 b 变成了 3x4 的二维数组。

```
In: b.shape = (3, 4)    # b 的形状被改变
In: b.resize((3, 4))    # 效果同上
```

转置矩阵是线性代数中很重要的操作，NumPy 提供了用于矩阵转置的 transpose()函数。

```
In: c.transpose()     # 矩阵转置，此处也可写为 c.T
Out:
array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])
```

如果需要将一维数组调整为二维数组，那么可在数组对象中使用 np.newaxis 关键字。

```
In: ar = np.array([3, 4, 5])
In: ar.shape
Out: (3,)                          # 一维数组
In: ar = ar[ :, np.newaxis]        # 增加一个维度
In: ar.shape
Out: (3, 1)                        # 变为二维数组
```

多维数组也可调用 ravel()或 flatten()函数，以便展平为一维数组。

```
In: d1 = b.ravel()                 # 返回一维数组视图
In: d1
Out: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
In: d2 = b.flatten()               # 返回一维新数组（将分配新内存）
```

这两个函数的内存机制是不同的。ravel()返回原数组的视图，所以它和原数组共用内存；flatten()会分配新的内存来保存新数组，因而它和原数组是独立的。

NumPy 的 append()、insert()、delete()等函数可以实现数组元素的插入、删除等操作，示例如下。

```
In: b = np.array([1, 3, 5, 7])
In: np.append(b, [9, 10])          # append 在数组末尾插入新数据 9,10
Out: array([ 1,  3,  5,  7,  9, 10])

In: np.insert(b, 2, 20)            # insert 在数组 b 的第 2 个位置插入新数据 20
Out: array([ 1,  3, 20,  5,  7])

In: np.delete(b, [0, 1])           # delete 按照下标删除数组 b 中的第 0,1 两个元素
Out: array([5, 7])
```

上述三个方法都不会直接改变原数组，而是返回一个新数组。要改变原数组，可将返回结果赋值给原数组，如 b = np.append(b, [9,10])。

7.5.2 数组组合

有时，出于数据处理的需要，需要组合数组。NumPy 数组有水平组合、垂直组合和深度组合等多种方式，涉及的函数有 hstack()、vstack()、dstack()、column_stack()、row_stack()和 concatenate()等。

```
In: a = np.arange(6).reshape(2, 3)    # 生成测试用数组 a 和 b，shape 都是 2x3 大小
In: a
Out:
array([[0, 1, 2],
       [3, 4, 5]])
In: b = np.arange(10, 16).reshape(2, 3)
```

```
In: b
Out:
array([[10, 11, 12],
       [13, 14, 15]])
```

（1）水平组合：多个数组沿水平方向组合，使用 hstack()函数，如下所示。

```
In: np.hstack((a, b))                    # 水平组合。行数不变，列数增加
Out:
array([[ 0,  1,  2, 10, 11, 12],
       [ 3,  4,  5, 13, 14, 15]])
```

也可以用 concatenate()或 column_stack()函数实现同样的效果，如下所示。

```
In: np.concatenate((a, b), axis=1)   # axis=1 表示按第 1 轴，即水平横向
In: np.column_stack((a, b))          # 效果同上
```

（2）垂直组合：多个数组沿垂直方向组合，使用 vstack()函数，如下所示。

```
In: np.vstack((a, b))                # 垂直组合。行数增加，列数不变
Out:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [10, 11, 12],
       [13, 14, 15]])
```

也可以用 concatenate()或 row_stack()函数实现垂直组合。

```
In: np.concatenate((a, b), axis=0)   # axis=0 表示按第 0 轴，即垂直纵向
In: np.row_stack((a, b))             # 效果同上
```

（3）深度组合：多个数组按深度方向组合，使用 dstack()函数，如下所示。两个二维数组深度组合将构成一个三维数组。

```
In: c = np.dstack((a, b))
In: c
Out:
array([[[ 0, 10],
        [ 1, 11],
        [ 2, 12]],
       [[ 3, 13],
        [ 4, 14],
        [ 5, 15]]])
# 深度组合后，c 是一个三维数组。在 a，b 原有的第 0，1 轴后面增加了第 2 轴
In: c.shape
Out: (2, 3, 2)
```

从下面的例子可以看出，vstack()、hstack()、dstack()是分别沿第 0、1、2 轴进行组合的。

```
In: np.vstack((c, c)).shape   # vstack 沿第 0 轴组合
Out: (4, 3, 2)
In: np.hstack((c, c)).shape   # hstack 沿第 1 轴组合
Out: (2, 6, 2)
In: np.dstack((c, c)).shape   # dstack 沿第 2 轴组合
Out: (2, 3, 4)
```

7.5.3　数组分割

NumPy 提供 hsplit()、vsplit()、dsplit()等函数实现对数组的水平、垂直或深度分割。

可以将数组分割成相同大小的子数组，也可以直接指定分割的具体位置。注意，分割后得到的数组是视图，而不是复制的新数组。

（1）水平分割：数组沿水平方向分割，使用 hsplit()函数。

```
In: b = np.arange(24).reshape(4, 6)  # 生成 4x6 数组
In: b
Out:
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])
In: x1, x2 = np.hsplit(b, 2)         # 沿水平方向分为 2 个相同大小的 4x3 数组
In: x1
Out:
array([[ 0,  1,  2],
       [ 6,  7,  8],
       [12, 13, 14],
       [18, 19, 20]])
In: x1, x2, x3 = np.hsplit(b, 3)     # 分为 3 个相同大小的 4x2 数组
In: x1.shape, x2.shape, x3.shape
Out: ((4, 2), (4, 2), (4, 2))
# 下例指定分割位置，在原数组的第 1，4 列拆分，得到 3 个数组
# x4 含原数组的第 0 列，x5 含原数组的第 1、2、3 列，x6 含原数组的第 4、5 列
In: x4, x5, x6 = np.hsplit(b, [1, 4])     # 在第 1，4 列分隔
In: x4.shape, x5.shape, x6.shape
Out: ((4, 1), (4, 3), (4, 2))
```

（2）垂直分割：数组沿垂直方向分割，使用 vsplit()函数。

```
In: y1, y2 = np.vsplit(b, 2)     # 沿垂直方向分为 2 个相同大小的 2x6 数组
In: y1
Out:
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
In: np.vsplit(b, 4)              # 沿垂直方向分为 4 个相同大小的 1x6 子数组
Out:
[array([[0, 1, 2, 3, 4, 5]]),
 array([[ 6, 7, 8, 9, 10, 11]]),
 array([[12, 13, 14, 15, 16, 17]]),
 array([[18, 19, 20, 21, 22, 23]])]
```

（3）深度分割：沿多维（维度>=3）数组的最后一个轴分割，使用 dsplit()函数。注意 dsplit()不能用于一维和二维数组。

```
In: b = np.arange(24).reshape(2, 3, 4)    # 生成 2x3x4 的三维数组
In: v1, v2 = np.dsplit(b, 2)              # 沿深度分为 2 个相同大小的 2x3x2 数组
In: v1.shape, v2.shape
Out: ((2, 3, 2), (2, 3, 2))
```

7.5.4 读写文件

数据处理时，我们通常要将大量数据保存到文件中。NumPy 提供读写文本文件和二进制文件

的两类函数，读写文本文件使用 savetxt()和 loadtxt()函数。

```
In: b = np.arange(95, 101).reshape(2, 3)
In: b
Out:
array([[ 95,  96,  97],
       [ 98,  99, 100]])
In: np.savetxt('data.txt', b)    # 将数组 b 保存到文件 data.txt 中
In: !type  data.txt              # 查看文件 data.txt，!type 是 IPython 支持的查看命令
9.50000000000000000e+01 9.60000000000000000e+01 9.70000000000000000e+01
9.80000000000000000e+01 9.90000000000000000e+01 1.00000000000000000e+02
```

由上例可见，由于 savetxt()存盘时默认采用科学记数法，因此数据 98 的保存形式类似于 9.80000000000000000e+01。要想保存为其他格式，可在 savetxt()函数中用 fmt 参数指定保存格式，fmt 的常用格式符如表 7.4 所示。

```
# 将数组 b 以 i（整数）格式保存为文件 data2.txt
# delimiter 参数指定以逗号分隔，fmt 指定格式
In: np.savetxt('data2.txt', b, delimiter=',' , fmt="%i")
In: !type  data2.txt     # 可见文件中的数据为整数格式
95,96,97
98,99,100
```

表 7.4　fmt 的常用格式字符

字　符	格式含义
c	单个字符
d 或 i	十进制有符号整数
e 或 E	科学记数法表示的浮点数
f	浮点数
s	字符串
u	十进制无符号整数
x 或 X	十六进制无符号整数

文本数据文件可使用 loadtxt()函数读取。

```
In: np.loadtxt('data2.txt', delimiter=',')
Out:
array([[ 95.,  96.,  97.],
       [ 98.,  99., 100.]])
```

除了文本文件，NumPy 还支持其特有的.npy 格式的二进制文件。这种文件读取速度更快，使用 save()和 load()函数存取。

```
In: np.save("data", b)           # 数组 b 保存为 data.npy 文件（自动加上扩展名.npy）
In: d = np.load('data.npy')      # 读取 npy 格式的二进制文件
In: d
Out:
array([[ 95.,  96.,  97.],
       [ 98.,  99., 100.]])
```

7.6 应用实例

本节用 NumPy 给出一个骰子研究实例。有三颗骰子，每次一起抛出，现在用随机函数模拟抛掷 1000 次。这 1000 次中有多少次投出“666”？有多少次出现三个骰子点数一样的情况？如果假定最初有 500 个筹码，三个骰子的点数之和大于 10 算赢，小于等于 10 算输，每次输赢一个筹码，那么最后是赢还是输？曾经达到的最大输赢数是多少？下面给出求解这个问题的完整代码，读者请留意代码中的注释。

```
import numpy as np
np.random.seed(7)                              # 注释该行则每次测试有变化
num = 1000                                     # 抛掷总次数
# 产生 1000x3 数组 dice，随机整数范围[1,6]，注意函数中要写为 7
dice = np.random.randint(1, 7, size=(num, 3))
# 为编程方便，计算三个骰子的点数之和，添加到 dice 末尾列，现在是 1000x4 数组
dice = np.column_stack((dice, dice.sum(axis=1)))

point = dice[:, 3]                                     # 取点数之和列
print('投掷出 666 的次数:', (point == 18).sum())     # 出现 666 的次数
print('666 出现在第 x 次', np.where(point == 18))    # 出现在第 x 次

# 找出三次投掷点数一样的行，得到一个布尔数组，注意 np.logical_and 函数
condition = np.logical_and(dice[:, 0] == dice[:, 1], dice[:, 1] == dice[:, 2])
print('\n 三次投掷点数均相同的次数:', condition.sum())   # 计算 True 的次数
print('投掷情况为:', dice[condition])                    # 显示 True 对应的行

money = 500                                            # 初始筹码数
bep = 10                                               # >10 赢，<=10 输
print('\n 初始筹码:', money, ', 盈亏点:', bep, ', 每次输赢 1')
# 只需简单统计> bep 和<= bep 的次数之和，即可计算最后的筹码数
total = money + (point > bep).sum() - (point <= bep).sum()
print('最后筹码:', total)

# 计算投掷过程中曾经达到的最高筹码数、最低筹码数
# np.where 判断每次输赢，返回 1,-1 构成的每次盈亏列，添加到末尾列，变为 1000x5
dice = np.column_stack((dice, np.where(dice[:, 3] > bep, 1, -1)))

# np.cumsum 累加每次盈亏列，得到累积盈亏列，添加到末尾列，变为 1000x6
dice = np.column_stack((dice, np.cumsum(dice[:, 4])))

# 累积盈亏列加初始筹码数，得到当前筹码列，添加到末尾列，变为 1000x7
dice = np.column_stack((dice, dice[:, 5] + money))

s = '曾经的最高筹码数{}, 出现在第{}次'
print(s.format(np.max(dice[:, 6]), np.argmax(dice[:, 6])))
s = '曾经的最低筹码数{}, 出现在第{}次'
print(s.format(np.min(dice[:, 6]), np.argmin(dice[:, 6])))
```

程序运行后，显示结果如下：

```
投掷出 666 的次数: 2
666 出现在第 x 次 (array([ 99, 865], dtype=int64),)

三次投掷点数均相同的次数: 22
投掷情况为: [[ 5  5  5 15]
             [ 5  5  5 15]
             [ 6  6  6 18]
               ...  ...
初始筹码: 500 , 盈亏点: 10 , 每次输赢 1
最后筹码: 442
曾经的最高筹码数 504，出现在第 57 次
曾经的最低筹码数 431，出现在第 810 次
```

图 7.1 直观地反映了筹码的变化，第 9 章中将介绍绘图。

```
import matplotlib.pyplot as plt                # 引入绘图库
plt.rcParams['font.sans-serif'] = ['SimHei']   # 指定中文黑体字体
plt.xticks(fontsize=14)                        # 设 x 轴文字大小
plt.yticks(fontsize=14)                        # 设 y 轴文字大小
plt.title('筹码变化', fontsize=16)              # 设标题文字
plt.plot(dice[:, -1])                          # 画筹码变化折线图
```

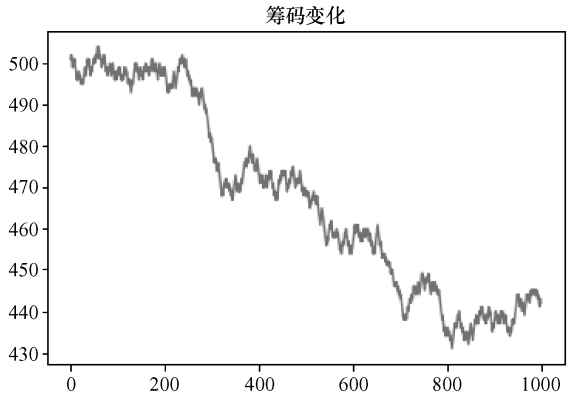


图 7.1　筹码变化图

程序运行中，dice 从最初的 1000×3 数组变为最后的 1000×7 数组，dice 数据如下。

```
In: np.set_printoptions(threshold=10000)
# 调高 threshold 以输出所有数据，中间列不用省略号代替
In: dice[:3]                                  # 显示前三行
Out: array([[  5,   2,   4,  11,   1,   1, 501],
            [  4,   5,   2,   11,  1,   2, 502],
            [  1,   2,   3,   6,  -1,   1, 501]])
In: dice[-3:]                                 # 显示后三行
Out:array([[  4,   6,   3,  13,   1, -58, 442],
           [  3,   5,   3,  11,   1, -57, 443],
           [  1,   6,   2,   9,  -1, -58, 442]])
```

使用 NumPy 编程时，程序员应充分利用 NumPy 的各种函数，避免使用 for 循环语句去处理批量数据，要将数组作为一个整体来运算。

7.7 本章小结

本章介绍了 Python 生态圈最重要的基础软件包 NumPy 及其提供的数组运算。数组的重要属性有维度 arr.ndim、元素个数 arr.size 和数组形状 arr.shape。数组元素可以整数索引或切片访问，也可布尔索引筛选访问。NumPy 的数组切片产生的是一个视图，视图和原数组指向同一内存空间，两者相互影响。布尔索引筛选的结果是一个复制的新数组而不是视图。

数组运算是整体进行的，无须编写循环程序处理。数组可以和单个数据或数组运算，形状不同的数组按广播规则计算。

NumPy 提供很多常用函数，如 sum()、min()、mean()、median()、max()、var()等。numpy.random 随机数模块含有常用随机数函数，如 randn()、randint()、choice()、seed()等。

NumPy 支持数学多项式运算，函数 poly1d()用于构造多项式，函数 polyfit()用于拟合多项式。

数组可用 reshape()方法改变维度。hstack()、vstack()、column_stack()、row_stack()等方法用于组合数组，hsplit()、vsplit()方法用于分割数组，savetxt()和 loadtxt()方法用于读写文本数据文件。

习题

1. 用 NumPy 和列表各生成 100000 个随机数据，完成各自的求和运算，比较运算时间［提示：可使用 time.perf_counter()函数记录时间］。
2. NumPy 数组切片和 Python 列表切片有何区别？
3. NumPy 数组有哪些重要特性？
4. NumPy 中有哪些常用的函数？
5. 改变数组的维度有哪些方法？
6. 试举一例，完成数组的广播计算。
7. 数组如何进行组合和拆分？
8. NumPy 如何存取文本数据文件？
9. 利用 NumPy 多项式解方程 $x^2 + x - 6 = 0$
10. 构造一个 50×4 的成绩数组，第 0 列代表学号 1～50，第 1～3 列代表三门成绩（成绩取区间[40, 100]内的随机整数），给出常规的成绩统计数据和排序数据。

第 8 章　Pandas 数据分析库

Pandas 是目前 Python 生态圈中最常用的数据分析工具库之一。该库以 NumPy 为基础，增加了标签支持，整合了对数据集的读取、清洗、转换、分析、统计、绘图等一系列工作流程，能够高效地处理和分析各类数据，其官网地址为 http://pandas.pydata.org/。

本章介绍 Pandas 的基本数据结构和常用操作，绘图操作将在第 9 章讲述。本章案例的工作环境同前一章，即仍然在 IPython 下操作。如果需要升级或重新安装 Anaconda 中的 Pandas，那么可在命令行执行如下命令：

```
conda  install  pandas                    # 安装 Pandas
conda  update  pandas                     # 升级 Pandas
```

在使用 Pandas 的各项功能之前，需要明确导入 Pandas 工具包。Python 社区约定按如下惯例导入 Pandas 包：

```
import  numpy  as  np
import  pandas  as  pd
from   pandas  import  DataFrame,  Series
```

后续示例中都以 pd 指代 Pandas。使用 Pandas 处理的数据表较大时，为更好地查看大数据表，可设置如下显示选项：

```
pd.set_option('display.max_columns', 10)  # 最多显示 10 列，中间用…代替
pd.set_option('display.width', 120)       # 每行最多显示的字符数，超过时折行
pd.set_option('display.max_rows', 30)     # 每个表最多显示 30 行
pd.set_option('precision', 2)             # 显示精度（小数点后 2 位）
```

8.1　Pandas 的基本数据结构

Pandas 提供了三种基本的数据结构：

- Series：带标签的一维数组
- DataFrame：带标签的二维数组（即表格）
- Panel：带标签的三维数组（可视为若干表格的叠加面板）

日常处理数据时主要使用 Series 和 DataFrame。与 NumPy 数组相比，这两种数据结构最重要的改进是增加了对标签（也称轴索引）的支持，可以实现自动的按索引对齐运算。

8.1.1　序列

序列（Series）类似 NumPy 的一维数组，它由一组数据项和标签（即索引）组成。numpy.ndarray 数组只能按整数下标索引，而 Series 允许对数据自定义标签并能基于标签对齐运算。

```
In: import  pandas  as  pd
In: from   pandas  import  DataFrame, Series
In: pd.__version__                # 显示 Pandas 的版本
Out: '0.24.2'
In: s1 = Series([10, 20, 30])     # 创建 Series，默认索引是整数下标
In: s1
```

```
Out:
0    10
1    20
2    30
dtype: int64
```

Series 也可以在创建时自定义标签索引，如下所示。

```
# index 参数定义标签，注意标签个数和数据个数必须一致
In: s2 = Series([10, 20, 30], index=['a', 'b', 'c'])
In: s2
Out:
a    10
b    20
c    30
dtype: int64

In: s2['a']        # 用标签取数据，此处也可写为 s2.a
Out: 10
In: s2[0]          # 有了自定义标签，仍可按整数索引取数据
Out: 10
In: s2['a':'c']    # 标签切片，注意标签切片时包含末尾的数据
Out:
a    10
b    20
c    30
dtype: int64
In: s2[0:2]        # 整数索引切片，不包含末尾的数据
Out:
a    10
b    20
dtype: int64
```

自定义标签后，就不用再记忆整数下标了，因此方便了用户记忆和引用。因为 Pandas 是基于 NumPy 构建的，所以 NumPy 数组具有的性质和运算 Series 都支持。

```
In: s2.sum(), s2.mean(), s2.median()      # 求和、均值、中位数
Out: (60, 20.0, 20.0)
In: s2['a'] = 100                         # 修改'a'标签对应的值
In: s2['d'] = 200                         # 'd'标签不存在时，将创建此标签
In: s2
Out:
a    100
b     20
c     30
d    200
dtype: int64
```

Series 的用法类似于字典类型的用法。执行“obj['标签']=新值”时，若标签已存在则修改对应值，若标签不存在则添加标签并赋值。

Series 支持多种创建对象的方法，如下所示。

```
In: s0 = Series(np.random.rand(4))            # 以 NumPy 数组为参数创建
In: s1 = Series([1, 2, 3], index=list('axb')) # 以列表为参数并指定标签
In: s1
Out:
a    1
x    2
b    3
dtype: int64

In: s2 = Series({'a':10, 'b':20, 'c':30})     # 以字典为参数创建
In: s2                                        # 字典的键自动成为标签
Out:
a    10
b    20
c    30
dtype: int64

In: s1 + s2     # 标签将自动对齐运算，不匹配的标签运算结果为 NaN
Out:
a    11.0
b    23.0
c    NaN
x    NaN
dtype: float64
```

Series 具有标签，数据在运算时会自动对齐标签进行计算，这一特性也是 Pandas 相比 NumPy 的重要改进。上例中，s1 的标签为['a','x','b']，s2 的标签为['a','b','c']，执行 s1 + s2 运算时，两者都具有的'a'、'b'标签自动对齐，对应的数值相加；两者互不匹配的'c'、'x'标签相加后的值为 NaN。NaN(not a number)是 Pandas 中表示非数值或缺失值的符号。

Series 对象具有 values 和 index 属性，分别对应于其数据值和索引值。

```
In: s2.values                         # 数据
Out: array([10, 20, 30], dtype=int64)
In: s2.index                          # 索引
Out: Index(['a', 'b', 'c'], dtype='object')
In: s2.index.value_counts()           # 统计每个索引出现的次数
Out:
b    1
a    1
c    1
dtype: int64
```

索引标签不能单个地修改，类似 s2.index[0]='d'的语句将引发“Index does not support mutable operations”错误，但可以对标签一次性全部赋值进行修改。

```
In: s2.index = ['d', 'e', 'f']    # 修改标签
In: s2
Out:
d    10
e    20
```

```
f    30
dtype: int64
```

Series 对象本身及其索引都有一个 name 属性，该属性在 Pandas 的一些操作中会被引用。

```
In: s2.name = 'data'        # 设置数据的 name 属性
In: s2.index.name = 'flag'  # 设置索引的 name
In: s2
Out:
flag                        # flag 是索引的 name
a    10
b    20
c    30
Name: data, dtype: int64    # data 是数据的 name
```

Series 对象为数据统计提供一些重要的属性和方法，如表 8.1 所示。

表 8.1 Series 对象的常用属性和方法

属性或方法名	功能描述
s.size	返回数据个数（注：此为属性，调用时不需要括号）
s.count()	返回非空数据的个数
s.unique()	返回不重复的数据值
s.value_counts()	统计每个数据值的出现次数
s.head(n)	只显示头部的 n 个数据
s.tail(n)	只显示尾部的 n 个数据
s.take([i1,i2,…])	按指定的索引下标取数据

```
In: s = Series([0, 1, 1, 3, 3, 3, np.nan, np.nan]) # 构造 Series，含 2 个 np.nan 值
In: s.size, s.count()
Out: (8, 6)                                # 共 8 个数据，有 6 个非空数据
In: s.unique()                             # 返回不重复的数据值
Out: array([ 0.,  1.,  3., nan])
In: s.value_counts()                       # 统计每个数据值的次数
Out:                                       # 表明序列中有 3 个 3.0，2 个 1.0，1 个 0
3.0    3
1.0    2
0.0    1
dtype: int64
In: s.take([1, 3, 5])                      # 取出索引号 1，3，5 的数据
Out:
1    1.0
3    3.0
5    3.0
```

8.1.2 数据框

DataFrame（数据框）是 Pandas 最重要的数据结构之一。数据框可视为一个由行和列构成的二维表格，每行或每列都可视为一个 Series。DataFrame 既有行索引又有列索引，因此也可以将数据框视为由 Series 组成的字典。数据框具有丰富的方法，可以非常方便地读取、清洗、统计数据并作图。

```
In: data = {'apple': [1100,1050,1200], 'huawei': [1250,1300,1328], 'oppo': [800,850,750]}
# 用字典 data 来创建数据框，字典的键自动成为列名
In: df = DataFrame(data, index=['一月',  '二月',  '三月'])   # index 标签
In: df
Out:
     apple  huawei  oppo
一月   1100    1250   800
二月   1050    1300   850
三月   1200    1328   750
In: df['apple']       # 访问一列，也可写为 df.apple
Out:
一月    1100
二月    1050
三月    1200
Name: apple, dtype: int64

In: df.loc['一月']    # 访问一行。注意语法，不能写为 df['一月']
Out:
apple     1100
huawei    1250
oppo       800
Name: 一月, dtype: int64
```

数据框的一行或一列数据即是一个 Series，所以操作单行或单列时可以按 Series 处理。

```
In: type(df.loc['一月']), type(df.apple)       # 查看行、列的数据类型
Out: (pandas.core.series.Series, pandas.core.series.Series)

# 将 df 保存为 mobile.csv 数据文件，GBK 是 Windows 的中文编码字符集
In: df.to_csv('mobile.csv', encoding='GBK')    # GBK 编码也可写为 cp936 编码
In: !type  mobile.csv                          # 显示数据文件内容
,apple,huawei,oppo
一月,1100,1250,800
二月,1050,1300,850
三月,1200,1328,750
```

上例中如不指定字符编码，Python 在保存时默认使用 utf-8 编码。当用!type 命令显示文件内容时，Windows 系统用 GBK 编码解析，中文将显示为乱码，所以在涉及中文信息保存或读取时应指定 encoding='GBK'参数。

```
# 将 mobile.csv 读入数据框 df2，index_col=0 指定文件第 0 列作为标签
In: df2 = pd.read_csv('mobile.csv', index_col=0, encoding='GBK')
```

创建 DataFrame 的另一种常见形式是使用嵌套字典参数，外层字典的键作为列名，内层的键作为行索引，如下所示。

```
In: df3 = DataFrame({'apple':  {'一月':1100, '二月':1050, '三月':1200},\
                     'huawei':  {'一月':1250, '二月':1300, '三月':1328},\
                     'oppo'  :  {'一月':800,  '二月':850,  '三月':750}})
```

数据框可视为二维表格，有 index（行）和 columns（列）两个重要属性。在创建数据框时，可以指定 index 和 columns。

```
In: frame=DataFrame(np.arange(12).reshape(3, 4), index=['a','b','c'], columns=['c1', 'c2','c3','c4'])
```

```
In: frame
Out:
   c1  c2  c3  c4
a   0   1   2   3
b   4   5   6   7
c   8   9  10  11
In: frame.columns                              # 显示列名
Out: Index(['c1', 'c2', 'c3', 'c4'], dtype='object')

In: frame.index                                # 显示行索引
Out: Index(['a', 'b', 'c'], dtype='object')
In: frame.index = [0, 1, 2]                    # 修改行索引
In: frame.columns = ['x1', 'x2', 'x3', 'x4']   # 修改列名
In: frame
Out:
   x1  x2  x3  x4
0   0   1   2   3
1   4   5   6   7
2   8   9  10  11

In: frame.describe()
Out:
         x1   x2    x3   x4
count    3.0  3.0   3.0   3.0
mean     4.0  5.0   6.0   7.0
std      4.0  4.0   4.0   4.0
min      0.0  1.0   2.0   3.0
25%      2.0  3.0   4.0   5.0
50%      4.0  5.0   6.0   7.0
75%      6.0  7.0   8.0   9.0
max      8.0  9.0  10.0  11.0
```

利用 describe()方法可快速得到数据框中每列数据的数据个数（count）、均值（mean）、标准差（std）、最小值（min）、最大值（max），以及 25%、50%、75%数据分位点的对应值等。

8.2 访问数据

为方便灵活地存取 DataFrame 数据，Pandas 提供了多种表达语法。下面先构造一个示例数据框。

```
In: df=DataFrame(np.arange(12).reshape(3,4),index=['a','b','c'], columns=['c1','c2', 'c3','c4'])
In: df
Out:
   c1  c2  c3  c4
a   0   1   2   3
b   4   5   6   7
c   8   9  10  11
```

可以用 df.info()、df.head(n)、df.tail(n)命令快速查看数据框中的数据。

```
In: df.info()                          # 显示表的基本信息
<class 'pandas.core.frame.DataFrame'>
Index: 3 entries, a to c
```

```
Data columns (total 4 columns):
c1    3 non-null int32
c2    3 non-null int32
c3    3 non-null int32
c4    3 non-null int32
dtypes: int32(4)

In: df.head(2)                          # 显示头两行数据
Out:
   c1  c2  c3  c4
a   0   1   2   3
b   4   5   6   7
In: df.tail(2)                          # 显示末尾两行数据
Out:
   c1  c2  c3  c4
b   4   5   6   7
c   8   9  10  11
```

8.2.1　loc[]、iloc[]访问

1. 按列访问

访问列数据时使用 object['列名']或 object.列名的形式。

```
In: df['c1']                        # 访问 c1 列数据，也可写为 df.c1
Out:
a    0
b    4
c    8
Name: c1, dtype: int32
In: df.c1.b                         # 访问 c1 列且行标签为 b 的数据，也可写为 df.c1['b']
Out: 4

In: df[['c1', 'c3']]                # 访问 c1，c3 两列，注意多个列名要放入[]
Out:
   c1  c3
a   0   2
b   4   6
c   8  10

In: df[df.c1 > 3]                   # 按条件访问，只显示 c1 列>3 的行
Out:
   c1  c2  c3  c4
b   4   5   6   7
c   8   9  10  11

In: df[(df.c1 > 3) & (df.c2 > 5)]     # 按条件访问，c1 列>3 且 c2 列>5
Out:
   c1  c2  c3  c4
c   8   9  10  11
```

[]内的逻辑运算符要使用 &（与）、|（或）、~（非），而不能使用 Python 的 and、or、not 运算符。同时注意，[]内的逻辑表达式要用小括号括起来，如上例中的(df.c1>3)和(df.c2>5)。

2. loc 和 iloc 存取器

访问行或块数据时常用 loc[]和 iloc[]存取器，其中 loc 基于标签，iloc 基于整数索引下标。loc[]中可以是[单行]、[单行，单列]、[行切片]、[行切片，列切片]等多种表达形式。loc[]和 iloc[]格式示例如表 8.2 所示。

表 8.2 loc[]和 iloc[]格式示例表

格式示例	说 明
df.loc['a']	访问行标签'a'对应的一行
df.iloc[0]	访问第 0 行
df.loc['a': 'c']	访问行标签'a'到'c'，共 3 行
df.iloc[0:2]	访问第 0，1 行，共 2 行
df.iloc[[0, 2]]	访问第 0，2 两行
df.iloc[0, 2]	访问第 0 行、第 2 列处的单个数据
df.iloc[: , 1]	访问第 1 列
df.iloc[: , [0,2]]	访问第 0，2 两列数据
df.iloc[:2, :2]	访问第 0～1 行和第 0～1 列交叉处的数据块

```
In: df.loc['a']        # loc 基于标签，取行标签'a'对应的行，得到一个 Series
Out:
c1    0
c2    1
c3    2
c4    3
Name: a, dtype: int32

In: df.iloc[0:2]      # iloc 基于整数下标，取第 0、1（不含第 2）行数据
Out:
   c1  c2  c3  c4
a   0   1   2   3
b   4   5   6   7
```

由于 loc[]基于标签而 iloc[]基于整数下标，所以不能出现 loc[1]、iloc['a']这样的写法。另外，loc['a':'c']将包含结尾的'c'标签，而 iloc[0:2]不包含结尾的第 2 行。

```
In: df.loc['b', 'c1']           # 访问 b 行 c1 列的数据，是单个值
Out: 4

In: df.loc['a':'b']             # loc 基于标签，取'a':'b'对应的数据行
Out:
   c1  c2  c3  c4
a   0   1   2   3
b   4   5   6   7

In: df.loc['a':'b', 'c1':'c2']  # 取数据块[行标签:列标签]
Out:
```

```
   c1  c2
a   0   1
b   4   5
In: df.loc[:, 'c2':'c3']        # 取数据块[行标签:列标签]
Out:
   c2  c3
a   1   2
b   5   6
c   9  10
```

旧版 Pandas 还支持一种可混合使用标签和整数索引的 ix[]取数器，但因易引起误解而不提倡使用。

8.2.2　at[]、iat[]、query()访问

1．at[]和 iat[]取数器

要取出单个数据，可以使用 at[]或 iat[]取数器。at[]基于标签，iat[]基于整数索引，格式均为[单行,单列]，取出的是单个数据而不是 Series。

```
In: df.iat[1, 2]            # iat 基于行/列的整数索引，行/列都从 0 开始编号
Out: 6                      # 第 1 行第 2 列的值是 6

In: df.at['b', 'c3']        # at 基于行/列的标签
Out: 6
```

2．query()查询取数

Pandas 对象支持 query()查询，它类似于数据库中的查询条件表达，根据列名条件进行数据筛选，返回 DataFrame 对象。

```
In: df
Out:
   c1  c2  c3  c4
a   0   1   2   3
b   4   5   6   7
c   8   9  10  11
In: df.query('c1 > 2')              # 查询 c1>2
Out:
   c1  c2  c3  c4
b   4   5   6   7
c   8   9  10  11
In: df.query('c1 > 2 and c2 > 6')   # 查询 c1>2 and c2>6
Out:
   c1  c2  c3  c4
c   8   9  10  11
```

8.3　算术运算和对齐

8.3.1　nan 缺失值处理

在数据处理中，有时得到的原始数据中含有缺失值。NumPy 用 np.nan 表示缺失值，Pandas 用

NaN(Not a Number)表示缺失值，Python 内置的 None 也视为缺失值。

Pandas 是基于 NumPy 构建的，自然也支持数组运算，但两者对 nan 值的处理有所不同。在 NumPy 中，运算时若有 np.nan 则默认返回 nan。在 Pandas 中，运算时若有 np.nan 则忽略 nan，用其他非 nan 数据进行运算并返回结果。

```
# 演示 NumPy 和 Pandas 对缺失值 nan 的处理的不同
In: b = np.array([1, 5, np.nan, np.nan, 10])
In: b
Out: array([ 1.,  5., nan, nan, 10.])
In: b.sum(), b.mean()       # 有 nan 值时，NumPy 运算返回 nan
Out: (nan, nan)

In: s=Series(b)             # 生成 Series，含有 nan
In: s
Out:
0     1.0
1     5.0
2     NaN
3     NaN
4    10.0
dtype: float64

In: s.sum(), s.mean()       # Series 默认忽略 nan 值
Out: (16.0, 5.333333333333333)
In: s.mean(skipna=False)    # False 表示不能忽略 nan，此时和 NumPy 相同，也返回 nan
Out: nan
```

对于 nan 缺失值的处理，Pandas 主要有如下几个函数。

（1）isnull()和 notnull()函数：判断每个值是否是 nan 值，返回布尔数组。

```
In: s
Out:
0     1.0
1     5.0
2     NaN
3     NaN
4    10.0
dtype: float64

In: s.isnull()          # 判断是否是 nan 值，是 nan 值时返回 True
Out:
0    False
1    False
2     True
3     True
4    False
dtype: bool
In: s.isnull().sum()    # 统计缺失值个数
Out: 2
In: s.notnull()         # 判断是否是非 nan 值，是非 nan 值时返回 True
```

```
Out:
0     True
1     True
2    False
3    False
4     True
dtype: bool
```

（2）dropna()函数：返回删除 nan 值后的新数据框。

```
In: df = pd.DataFrame(np.arange(12).reshape(4, 3), index=list('abcd'), columns= list('xyz'))
In: df.iloc[1:3, 0:2] = np.nan   # 特意设置几个 nan 值
In: df.iloc[2, 2] = np.nan
In: df
Out:
    x      y     z
a  0.0    1.0   2.0
b  NaN    NaN   5.0
c  NaN    NaN   NaN
d  9.0   10.0  11.0

In: df.dropna()                  # 含有 nan 值的行被删除
Out:
     x     y   z
a  0.0   1.0   2.0
d  9.0  10.0  11.0

In: df.dropna(axis=1)            # 按 axis=1 纵向判断，含有 nan 值的列被删除
Out:
Empty DataFrame
Columns: []                      # 列被删空
Index: [a, b, c, d]

In: df.dropna(how='all')         # all 表示某行的所有数据都为 nan 才删除
Out:
    x     y     z
a  0.0   1.0   2.0
b  NaN   NaN   5.0
d  9.0  10.0  11.0

In: df.dropna(thresh=2)          # 某行的 nan 值个数>=2 才删除
Out:
     x     y     z
a  0.0   1.0   2.0
d  9.0  10.0  11.0
```

（3）fillna()函数：将 nan 值用特定的值填充。

```
In: df.fillna(0)                      # 缺失值都用 0 填充
In: df.fillna(method='ffill')         # 缺失值用其前面的非 nan 值填充
Out:
```

```
     x     y     z
a  0.0   1.0   2.0
b  0.0   1.0   5.0
c  0.0   1.0   5.0
d  9.0  10.0  11.0

In: df.fillna(method='bfill')        # 缺失值用其后面的非 nan 值填充
Out:
     x     y     z
a  0.0   1.0   2.0
b  9.0  10.0   5.0
c  9.0  10.0  11.0
d  9.0  10.0  11.0

In: df.fillna(value=df.mean())       # 用计算得到的平均值填充
Out:
     x     y     z
a  0.0   1.0   2.0
b  4.5   5.5   5.0
c  4.5   5.5   6.0
d  9.0  10.0  11.0
```

Pandas 还提供很多填充缺失值的方法（如插值填充），有兴趣的读者可从其官网下载并查看详细的电子文档。

8.3.2 对齐处理

Pandas 支持标签访问数据，运算时会自动基于标签对齐进行计算。运算数据若是 Series 则只有行标签，若是 DataFrame 则会在行、列两个方向上将数据的标签对齐，对应的数据元素进行计算，标签不匹配的数据元素默认标记为 NaN 值。

```
In: df = pd.DataFrame(np.arange(9).reshape(3, 3), index=list('abc'), columns= list('xyz'))
In: df
Out:
   x  y  z
a  0  1  2
b  3  4  5
c  6  7  8

In: s = pd.Series([1, 2, 3], index=list('xyz'))
In: s
Out:
x    1
y    2
z    3
dtype: int64

In: df + s            # DataFrame + Series，df 的列标签匹配 Series 行标签
Out:
   x  y   z
```

```
a  1  3   5
b  4  6   8
c  7  9  11
```

DataFrame 和 Series 之间的运算默认将 Series 的索引匹配到 DataFrame 的列，然后沿着行方向一直向下广播。如果某个索引值在 DataFrame 的列或 Series 的索引中找不到，那么参与运算的两个对象就会被重新索引以形成并集，如下所示。

```
In: s1 = pd.Series([1, 2, 3], index=list('wxy'))
In: s1
Out:
w    1
x    2
y    3
dtype: int64

In: df + s1                # 结果的列是 df 和 s1 列的并集[w,x,y,z]
Out:
    w    x     y   z
a NaN  2.0   4.0 NaN
b NaN  5.0   7.0 NaN
c NaN  8.0  10.0 NaN
```

两个数据框运算时，在行、列索引上都要对齐，如下所示。

```
In: df2 = pd.DataFrame(np.arange(9,18).reshape(3,3), index=list('abd'), columns= list('wxy'))
In: df2
Out:
    w   x   y
a   9  10  11
b  12  13  14
d  15  16  17

In: df + df2
Out:
    w     x     y   z
a NaN  10.0  12.0   NaN
b NaN  16.0  18.0   NaN
c NaN   NaN   NaN   NaN
d NaN   NaN   NaN   NaN
```

上例中 df 和 df2 在行标签上只有'a'、'b'匹配，列标签上只有'x'、'y'匹配，所以只有对应位置上的 4 个数据值相加，其余位置上都为 NaN。结果的行、列总数是两个数据集的并集。

存在缺失值时，可指定一个特殊的填充值 fill_value 替换缺失值。下例中的 df 和 df2 相加后，行索引和列索引是两者的并集。相加时某个位置上若一方是缺失值，则用 0 代替，例如 a 行 w 列。如某个位置上两方都是缺失值，则仍以 NaN 表示，例如 c 行 w 列。

```
In: df.add(df2, fill_value=0)  # 缺失的 NaN 值用 0 替代
Out:
      w     x     y    z
a   9.0  10.0  12.0  2.0
b  12.0  16.0  18.0  5.0
```

```
c  NaN   6.0   7.0  8.0
d  15.0  16.0  17.0  NaN
```

8.3.3 通用函数

由于 Pandas 是基于 NumPy 构建的，所以 NumPy 支持的通用函数都可以在 Pandas 中使用。

```
In: df = pd.DataFrame(np.arange(9).reshape(3,3), index=list('abc'), columns= list('xyz'))
In:df
Out:
   x  y  z
a  0  1  2
b  3  4  5
c  6  7  8
In: df.sum()                # 默认 axis=0，按列求和。本例中也可写为 np.sum(df)
Out:
x     9
y    12
z    15
dtype: int64

In: df.sum(axis=1)          # axis=1，按行求和。本例中也可写为 np.sum(df, axis=1)
Out:
a     3
b    12
c    21
dtype: int64

In: df.sum().sum()          # 对所有数据求和
Out: 36
```

另一类常见的操作是将自定义的函数应用到各行或各列上，以对整行或整列进行统计。DataFrame 的 apply()方法可以实现这一功能。

```
In: f = lambda  x: x.max() - x.min()      # 定义函数 f，参数 x 将代表整行、整列
In: df.apply(f)                           # 在每列上求最大值-最小值
Out:
x    6
y    6
z    6
dtype: int64
In: df.apply(f, axis=1)                   # 在每行上求最大值-最小值
Out:
a    2
b    2
c    2
dtype: int64

In: df.apply(lambda  x:x-x.mean())        # 计算每列数据与均值的差
Out:
      x    y    z
```

```
a   -3.0  -3.0  -3.0
b    0.0  0.0   0.0
c    3.0  3.0   3.0
```

apply()方法应用到整行、整列上，Pandas 还有一个 applymap()方法应用到单个数据上。下例定义了一个匿名函数 lambda x:str(x)*2，其中的 x 代表每个数据，str(x)将其转换为字符串，然后字符串乘以 2，所以返回的结果如下：

```
In: df.applymap(lambda  x:str(x)*2)
Out:
    x   y   z
a  00  11  22
b  33  44  55
c  66  77  88
```

8.4　读/写数据文件

Pandas 提供了多种数据文件接口，利用这些接口可以读取 CSV、Excel、JSON、HDF5 等格式的文件。

8.4.1　读/写 CSV 文件

CSV 文件是以逗号分隔的文本文件，常用作不同软件之间数据交换的中间文件。Pandas 提供 read_csv()和 to_csv()两个方法读/写 CSV 文件。

假定有 mobile.csv 文件（该文件已在 8.1.2 节创建），内容如下：

```
,apple, huawei, oppo
一月,1100,1250,800
二月,1050,1300,850
三月,1200,1328,750
```

```
In: df = pd.read_csv("mobile.csv", encoding='cp936', index_col=0)     # 读取文件
In: df
Out:
      apple  huawei  oppo
一月   1100    1250   800
二月   1050    1300   850
三月   1200    1328   750
```

读取 CSV 文件时使用了 read_csv()方法。文件 mobile.csv 中含有中文，保存时 Windows 系统默认采用 cp936 编码字符集，所以读取时也应指定该编码字符集。如果不指定为 cp936 编码字符集，那么 Pandas 默认按 utf-8 编码读取，这时会产生解码错误，提示'utf-8'编码解析错误。index_col=0 指定将文件的第 0 列数据作为索引标签使用。读取时 mobile.csv 文件的第 0 行被自动解析为列名。read_csv()有很多参数，可使用 pd.read_csv?命令查看详细的参数说明。

有些文件只含有数据，而没有列名、标签等内容。例如，数据文件 m2.csv 的内容如下：

```
1100,1250,800
1050,1300,850
1200,1328,750
```

```
# 文件中不含列名，可以在读取时用 names 参数自行指定列名
# 注：这个文件不含中文，只有数字，所以可不指定 encoding
```

```
In: df2 = pd.read_csv("m2.csv", names=['apple', 'huawei', 'oppo'])

# 也可以第一步：先读取
In: df3 = pd.read_csv("m2.csv", header=None)
# 第二步：再修改 columns 和 index
In: df3.columns = ['apple', 'huawei', 'oppo']
In: df3.index = ['一月', '二月', '三月']
```

上例中 header=None 表示文件不含列名。若不指定这个参数，则文件第 0 行“1100,1250,800”将被错误解析为列名。

有些数据文件不是用逗号分隔的，而是用空格分隔的。例如，数据文件 m4.txt 的内容如下：

```
1100   1250      800
1050   1300   850
1200  1328  750
```

```
# 读取空格分隔的数据，注意 sep="\s+"
In: df4 = pd.read_csv("m4.txt", sep="\s+", header=None)
```

因为 m4.txt 文件中的各个数据不是用逗号分隔的，而是用数量不等的空格或制表符分隔的，所以可以指定 sep="\s+"参数。"\s+"是正则表达式，表示分隔符可为若干空白字符。此例中若不指定 sep 参数，则读取的数据无法正确分解。

如果文件头部含有一些不需要读取的行，那么可指定 skiprows 参数将这些行跳过去。

```
pd.read_csv("数据文件名", skiprows=[0, 2])              # 跳过第 0, 2 行
pd.read_csv("数据文件名", skiprows=4 )                  # 跳过前 4 行
```

如果文件尾部含有一些不需要读取的行，那么可指定 skipfooter 参数将尾行跳过去。

```
pd.read_csv("数据文件名", skipfooter=2, engine = 'python')   # 跳过尾部的 2 行
```

数据文件很大，不想一次读取全部数据，而只想读取少数几行数据进行观察时，可指定 nrows 参数。

```
pd.read_csv("数据文件名", nrows=10)                     # 只读取前 10 行数据
```

假定有一个数据文件 stock.txt，其内容如下：

```
交易日          开盘       最高       最低       收盘         成交量
2019/03/22      18.09     18.63     18.02     18.15        43760812
2019/03/23      18.16     18.35     18.06     18.13        27830796
2019/03/24      18.11     18.11     17.68     17.72        27448272
```

该文件中包含了日期，在读取时指定参数 parse_dates=['交易日']，以便把该列解析为 Pandas 的日期格式，并用 index_col 参数指定为索引。文件内的各个数据是用空格或制表符分隔的，所以指定 sep='\s+'。读取该文件的命令如下。

```
In: df = pd.read_csv('stock.txt', parse_dates=['交易日'],encoding='cp936',
sep='\s+', index_col='交易日')
In: df
Out:
            开盘     最高    最低    收盘     成交量
交易日
2019-03-22  18.09  18.63  18.02  18.15  43760812
2019-03-23  18.16  18.35  18.06  18.13  27830796
2019-03-24  18.11  18.11  17.68  17.72  27448272

In: df.index          # 查看索引，可见为日期型索引
```

```
Out: DatetimeIndex(['2019-03-22', '2019-03-23', '2019-03-24'],
     dtype='datetime64[ns]', name='交易日', freq=None)
```

数据框对象自带 to_csv()方法，它可将数据存入 CSV 文件。

```
In: df.to_csv("d1.csv", encoding='cp936')              # 存盘，默认用逗号分隔
In: df.to_csv("d2.csv", encoding='cp936', sep=' ')     # 存盘时指定用空格分隔
```

8.4.2　读/写 Excel 文件

除 CSV 文件外，Excel 文件也是被广泛使用的一类文件。Pandas 提供了两个方法 read_excel()和 to_excel()读/写电子表格文件。假定有文件 mobile.xlsx，其内容如图 8.1 所示。

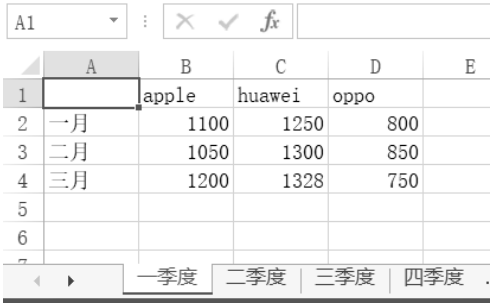


图 8.1　mobile.xlsx 文件

```
# 默认读第 1 个工作表，第 1 行作为列名，A 列（第 0 列）作为标签
In: df1 = pd.read_excel("mobile.xlsx", index_col=0)    # 读 Excel 文件

# sheet_name 指定读某个工作表
In: df2 = pd.read_excel("mobile.xlsx", sheet_name='二季度')
```

将数据框保存到 Excel 文件中时，使用 to_excel()方法。

```
In: df1.to_excel("a1.xlsx")                # 写入 Excel 文件
```

可以将多个数据框保存到一个 Excel 文件的不同工作表中。如下语句执行后，df1 和 df2 的数据分别保存在 test.xlsx 内名为"一季度"和"二季度"的工作表中。

```
from  pandas  import  ExcelWriter
with  ExcelWriter("test.xlsx")  as  writer:
    df1.to_excel(writer, sheet_name='一季度')
    df2.to_excel(writer, sheet_name='二季度')
```

8.4.3　读/写 HDF5 文件

HDF（Hierarchical Data Format）是一种用于存储和组织大量数据的文件格式，最初由美国国家超算中心研发，现在由非营利组织 HDF Group 支持。HDF 支持多种软件平台，包括 MATLAB、Java、Python、R 等。Pandas 也支持对 HDF5 文件的存取。

假定有两个数据框 df1 和 df2。将这些数据保存到 HDF5 文件内的命令如下。

```
# 若 store.h5 文件不存在，则先创建；若已存在，则打开此文件
In: store = pd.HDFStore('store.h5')
In: store['dfa'] = df1                 # 将 df1 保存到文件中，键名 dfa
In: store['dfb'] = df2                 # 将 df2 保存到文件中，键名 dfb
In: store.close()                      # 关闭文件
```

读出已保存的数据可以使用下面的命令。

```
In: store = pd.HDFStore('store.h5')  # 打开文件
In: df3 = store['dfa']               # 根据 dfa 键名取出数据
In: df4 = store['dfb']               # 根据 dfb 键名取出数据
```

从上面的示例可以看出，HDF 文件存取数据的方式类似于字典操作，都通过键存取数据。

8.5 数据整理

我们经常对已有的数据集做插入、删除、合并、排序等操作。Pandas 提供了很多数据整理方法，如表 8.3 所示。

表 8.3 Pandas 提供的数据整理方法

方 法 名	功能描述
df.drop(行标签)	返回删除行后的新数据框，原数据框不变
df.drop(列名, axis=1)	返回删除列后的新数据框，原数据框不变
df.pop(列名)	删除并返回某列，直接改变原数据框
df.insert(新列位置, 列名, 列值)	插入新列，直接改变原数据框
df.reindex(标签列表)	重索引（只保留指定的标签），原数据框不变
df.reset_index()	将索引列变为数据列
df.set_index(列名)	将数据列变为索引列
df.rename()	更改列名
df.append()	末尾插入
df.duplicated()	检测重复值，返回布尔数组
df.drop_duplicates()	删除重复值
pd.concat(df1, df2)	合并数据框

8.5.1 行、列的插入和删除

先准备测试用的数据框对象。

```
In: df = DataFrame({'姓名':['a','b'], '学号':['A1','A2'], '成绩 1':[98,90], '成绩 2':[87,80]})
In: df
Out:
  姓名   学号   成绩 1   成绩 2
0  a     A1     98       87
1  b     A2     90       80
```

1. 行的插入/删除

要在数据框末尾插入一行，可使用 append()方法，它以字典格式提供新行的数据。

```
# 在末尾插入新行，注意要指定 ignore_index=True 参数
In: df = df.append({'姓名':'d','学号':'A4','成绩 1':89,'成绩 2':78},ignore_index=True)
In: df
Out:
  姓名  学号  成绩 1  成绩 2
0  a     A1   98       87
1  b     A2   90       80
2  d     A4   89       78
```

若有结构相同的数据框 df 和 df2，则可用 df=df.append(df2)命令将两个数据框连接起来。

若要删除行，则可使用 drop()方法。DataFrame 的很多命令并不直接改变原数据框，而是返回新数据框，这和列表的处理方式不同。需要直接修改原数据框时，可将命令写为 df=df.append()的形式，或在命令中指定 inplace=True 参数。

```
In: df.drop(2, inplace=True)      # 按索引删除第 2 行，True 表示修改原数据框
```

上例由于直接修改了 df，因此删除了刚才插入的第 2 行，此时 df 中就只有两行数据。

2. 列的插入/删除

创建新列最简单的方法是直接给一个新列赋值，新列默认插在最后一列。要注意提供的数据个数应等于数据框的行数。

```
In: df['性别'] = ['M', 'F']       # 增加"性别"列，给新列赋值即可插入列
In: df
Out:
  姓名  学号  成绩 1  成绩 2 性别
0  a     A1   98       87     M
1  b     A2   90       80     F
```

要在数据框的中间位置插入新列，可以使用 insert()方法指定插入到第几列。

```
# 在第 4 列插入平均成绩，插入值由成绩 1 和成绩 2 计算得到
In: df.insert(4, '平均成绩', (df.成绩 1 + df.成绩 2) / 2 )
In: df
Out:
  姓名  学号  成绩 1  成绩 2  平均成绩  性别
0  a     A1   98       87       92.5    M
1  b     A2   90       80       85.0    F
```

删除列时，可使用如下三种方法。

```
In: df.drop('平均成绩', axis = 1, inplace=True)     # 注意 axis=1 和 inplace 参数
In: df.pop('成绩 1')
In: del  df['成绩 2']
In: df                                        # 删除后，df 只剩下 3 列
Out:
  姓名 学号 性别
0  a    A1   M
1  b    A2   F
```

8.5.2　索引整理

1. reindex()重建索引

reindex()方法用于行列的取舍。该方法保留指定标签的数据，抛弃未指定的标签。

```
In: df = pd.read_csv("mobile.csv", encoding='cp936', index_col=0)
In: df
Out:
       apple  huawei  oppo
一月   1100    1250   800
二月   1050    1300   850
三月   1200    1328   750

In: df2 = df.reindex(['一月', '二月', '四月'])          # 保留一、二月，新建四月
```

```
In: df2
Out:
      apple   huawei   oppo
一月  1100.0  1250.0   800.0
二月  1050.0  1300.0   850.0
四月  NaN     NaN      NaN
```

上例中新数据框 df2 只保留了一月和二月的数据，丢弃了三月的数据，同时建立了一个'四月'新标签，新标签对应的值默认为 NaN。

```
In: df.reindex(['apple', 'huawei','mi'], axis=1)        # axis=1 在列上重建索引
Out:
       apple  huawei   mi
一月   1100   1250     NaN
二月   1050   1300     NaN
三月   1200   1328     NaN
```

上例中在列上只保留了 apple 和 huawei，丢弃了 oppo，同时增加了一个新列 mi，新列的值为 NaN。

2. rename()重命名

已有的列名不太合适时，可以使用 rename()方法进行调整。

```
In: df.rename(columns={'apple':'Apple', 'huawei':'HW'})  # 更改列名
Out:
      Apple  HW    oppo
一月   1100  1250   800
二月   1050  1300   850
三月   1200  1328   750
```

更改列名后将返回一个新的数据框，而并不直接改动旧数据框。如果希望直接改动旧数据框，那么就要增加参数 inplace=True。Pandas 的很多命令都支持 inplace 参数。

```
In: df.rename(columns={'apple':'Apple', 'huawei':'HW'}, inplace=True)
```

3. set_index()重新设定索引列

有时，数据框的索引列不太合意。例如，学生表可能初始用整数序号作为索引，而我们想修改为用学号作为索引，此时可以使用 set_index()和 reset_index()方法。

```
In: df = DataFrame({'姓名':['a','b'], '学号':['A1','A2'], '成绩1':[98,90], '成绩2':[87,80]})
In: df
Out:
  姓名  学号  成绩1  成绩2
0  a     A1    98     87
1  b     A2    90     80

In: df3 = df.set_index('学号')  # 返回的新数据框将学号列设为索引
In: df3
Out:
    姓名  成绩1  成绩2
学号
A1   a     98     87
A2   b     90     80
```

如上所示，现在学号变为索引列。如果又想将姓名设为索引，那么可分如下两步处理。

```
In: df3.reset_index(inplace=True)          # 先将 df3 的原索引列学号恢复为数据列
In: df3
   学号 姓名  成绩 1  成绩 2
0   A1   a    98     87
1   A2   b    90     80
In: df3.set_index('姓名', inplace=True)   # 再将姓名列设为索引列
```

8.5.3　重复值处理

Pandas 提供了检测和删除重复值的方法，如下所示。

```
In: s = Series(list('abac'))
In: s
Out:
0    a
1    b
2    a
3    c

In: s.duplicated()      # 检测重复值，返回布尔数组，重复值处显示 True
Out:
0    False
1    False
2     True
3    False
dtype: bool

In: s.drop_duplicates()  # 删除重复值
Out:
0    a
1    b
3    c
```

对于数据框，在删除重复值时可指定判断的列。

```
In: df = DataFrame({'c1': ['a', 'a', 'b'], 'c2': ['a', 'b', 'b'], 'c3': ['a', 'b', 'x']})
In: df
Out:
  c1 c2  c3
0  a  a  a
1  a  b  b
2  b  b  x
In: df.drop_duplicates('c1')    # c1 列上删除重复值
Out:
  c1 c2 c3
0  a  a  a
2  b  b  x

In: df.drop_duplicates('c2')    # c2 列上删除重复值
Out:
```

```
  c1  c2 c3
0  a  a  a
1  a  b  b
```

8.5.4 排序和排名

1. 排序

数据集排序是一种常见的操作。排序一般可以按索引或数据值进行。对行或列索引排序时使用 sort_index()方法，按数据值排序时使用 sort_values()方法。排序后返回一个有序的新对象，而并不直接改变原数据集的顺序。要直接变动原数据集，可使用类似于 s=s.sort_index()或 s.sort_index(inplace=True)的语句。

```
In: s = Series([2, 5, 1], index=['d', 'a', 'b'])
In: s
Out:
d    2
a    5
b    1
dtype: int64

In: s.sort_index()   # 按索引'a b d'排序，返回新对象，并不改变原对象
Out:
a    5
b    1
d    2
dtype: int64

In: s.sort_values()  # 按数据值 1 2 5 排序
Out:
b    1
d    2
a    5
dtype: int64

In: s.sort_index(ascending=False)    # 按索引逆序排
Out:
d    2
b    1
a    5
dtype: int64
```

对数据框排序时，可以设定 axis 参数以指定按行或按列排序。

```
In: np.random.seed(7)                          # 设置随机数种子值 7
In: arr = np.array(np.random.randint(1, 100, size=9)).reshape(3, 3)
In: df = DataFrame(arr.reshape(3, 3), columns=['x','y','z'], index=['a','b','c'])
In: df
Out:
    x   y   z
a  48  69  26
```

```
b  68  84  24
c  93  58  15

In: df.sort_index(axis=1, ascending=False)     # 按列名索引降序 z y x 排列
Out:
    z   y   x
a  26  69  48
b  24  84  68
c  15  58  93

In: df.sort_values(by='y')                     # 按 y 列的数值排序
Out:
    x   y   z
c  93  58  15
a  48  69  26
b  68  84  24

In: df.sort_values(by=['y','z'])               # 先参照 y 列，再参照 z 列排序
```

注意，无论是升序排序还是逆序排序，缺失值都排在末尾。

2. 排名

排名和排序类似，但会自动生成一个排名号。排名操作使用 rank()方法。

```
In: s = Series([3, 5, 8, 5], index=list('abcd'))
In: s
Out:
a    3
b    5
c    8
d    5
dtype: int64

In: s.rank()   # 排名，默认按数据值升序排名
Out:
a    1.0
b    2.5
c    4.0
d    2.5
dtype: float64
```

上例中索引 a 的数值最小，排第 1。索引 b,d 的数值相同，应排在第 2、3 名，取平均名次(2+3)/2=2.5，索引 c 排在第 4。

```
In: s.rank(method='first')  # 指定名次号的生成方法为 first
Out:
a    1.0
b    2.0
c    4.0
d    3.0
dtype: float64
```

上例中名次号的生成方法为 method='first'，表示排名相同时不计算平均名次，而是以数据出现的先后顺序排列。要按数据值逆序排名，可使用 s.rank(ascending=False)命令。

8.5.5 数据框连接

Pandas 提供了 merge()方法，它可连接不同数据框的行，类似于关系数据库的连接运算，是多表处理中常见的操作。

```
In: df1 = DataFrame({'color':['r', 'b', 'w', 'w'], 'c1':range(4)})
In: df1
Out:
  color  c1
0     r   0
1     b   1
2     w   2
3     w   3

In: df2 = DataFrame({'color':['b', 'w', 'b'], 'c2':range(2, 5)})
In: df2
Out:
  color  c2
0     b   2
1     w   3
2     b   4

In: pd.merge(df1, df2)   # 或写为 pd.merge(df1, df2, on='color')
Out:
  color   c1  c2
0     b   1   2
1     b   1   4
2     w   2   3
3     w   3   3
```

df1 和 df2 有同名列 color，pd.merge()自动将同名列作为连接键，横向连接两个数据框的 color 键值相等的行。连接时，会丢弃原 dataframe 的索引。

两个数据框的列名不同时，用 left_on 和 right_on 参数分别指定。下例中指定 c1, c2 列为键，表示当 df1 表的 c1 列值等于 df2 表的 c2 列值时满足连接条件。因为 df1, df2 中的 color 列名相同，所以连接后自动加上后缀_x, _y 进行区分。

```
In: pd.merge(df1, df2, left_on='c1', right_on='c2')
Out:
   color_x   c1  color_y  c2
0       w    2      b     2
1       w    3      w     3
```

pd.merge 默认做 inner 内连接，其他连接方式有"left"、"right"和"outer"。这些方式和数据库中的左外连接、右外连接、外连接的运算规则是相同的。

```
In: pd.merge(df1, df2, how="left")    # 左外连接
In: pd.merge(df1, df2, how="right")   # 右外连接
In: pd.merge(df1, df2, how="outer")   # 外连接
```

```
Out:
  color  c1   c2
0     r   0  NaN
1     b   1  2.0
2     b   1  4.0
3     w   2  3.0
4     w   3  3.0
```

pd 的连接键位于索引中时，可指定参数将索引用于连接键，如下例所示。df1 中的索引值 0，1，2 分别和 df2 中的索引值 0，1，2 对应，连接匹配行得到结果。

```
In: pd.merge(df1, df2, left_index=True, right_index=True)    # 将索引用于连接键
Out:
   color_x  c1  color_y  c2
0       r   0       b   2
1       b   1       w   3
2       w   2       b   4
```

与连接有关的另一个方法是 pd.concat()，它合并两个数据框。

```
In: np.random.seed(7)
In: df1 = DataFrame(np.random.rand(4).reshape(2, 2), columns=['c1', 'c2'])
In: df2 = DataFrame(np.random.rand(4).reshape(2, 2), columns=['c1', 'c2'])
In: df1
Out:
         c1        c2
0  0.076308  0.779919
1  0.438409  0.723465
In:df2
Out:
        c1        c2
0  0.97799  0.538496
1  0.50112  0.072051

In: pd.concat([df1, df2], ignore_index=True)   # 默认沿纵向合并，行数增加
Out:
         c1        c2
0  0.076308  0.779919
1  0.438409  0.723465
2  0.977990  0.538496
3  0.501120  0.072051

In: pd.concat([df1, df2], axis=1)              # axis=1 沿横向合并，列数增加
Out:
         c1        c2      c1       c2
0  0.076308  0.779919  0.97799  0.538496
1  0.438409  0.723465  0.50112  0.072051
```

8.5.6　数据分段

数据分段是指将数据按指定的区间归类，以便统计每个区间的数据个数。例如，将年龄分为

老、中、青、儿童区间段，将成绩分为优、良、中、不及格区间段等。Pandas 中用于数据分段的方法是 pd.cut()，在分段之前要自定义数据区间段，还可以设置对应的标识文字。下面来看分段的例子。

```
In: np.random.seed(7)
In: score = np.random.randint(30, 100, size=100)   # 生成 100 个随机整数
In: bins = [0, 59, 70, 85, 100]                    # 定义区间段
In: labels = ['不及格', '中', '良', '优']           # 设置各段的标识文字
In: scut = pd.cut(score, bins, labels=labels)      # 将 score 按 bins 分段
In: type(scut)                                     # 查看 scut 的数据类型
Out: pandas.core.arrays.categorical.Categorical

In: scut
Out:
[良, 优, 不及格, 优, 不及格, ..., 良, 不及格, 不及格, 优, 优]
Length: 100
Categories (4, object): [不及格 < 中 < 良 < 优]

In: pd.value_counts(scut)                          # 统计各类别的数据个数
Out:
不及格  42
优      24
良      22
中      12
dtype: int64
```

从上例可见，分段后得到的 scut 对象类型为 Categorical。分段时将原始数据归入某个数据区间段，得到以标识文字表示的分段数据。分段的区间默认是右包含，即(0,59]对应“不及格”，(59,70]对应“中”，(70,85]对应“良”，(85,100]对应“优”。pd.value_counts()的统计结果是一个 Series，如下所示：

```
In: s = pd.value_counts(scut)        # 统计的结果是 Series
In: type(s)
Out: pandas.core.series.Series

In: s.index                          # 查看索引
CategoricalIndex(['不及格', '优', '良', '中'],
categories=['不及格', '中', '良', '优'], ordered=True, dtype='category')
```

在上条命令中查看索引可知，分段后['不及格', '中', '良', '优']形成了一个 category 数据类，自动构成了一个有序排列，所以用下面的命令按索引顺序排列。

```
In: s.sort_index()                   # 按索引升序排列
Out:
不及格  42
中      12
良      22
优      24
dtype: int64

In: s.sort_values(ascending=False)   # 按数据降序排列
```

```
Out:
不及格  42
优      24
良      22
中      12
dtype: int64
```

8.5.7　多级索引

Pandas不仅支持普通的一级索引，而且支持多级索引（MultiIndex）。多级索引可以更好地表达数据之间的联系。假定 A、B 两类产品都有红、绿两种颜色，构造销售数据如下。

```
In: mindex = pd.Index([('A', 'r'), ('A', 'g'), ('B', 'r'), ('B', 'g')], name=['product', 'color'])
# 创建多级索引
# 利用多级索引创建数据框
In:df = DataFrame(np.arange(2, 10).reshape(4, 2), index=mindex, columns=['一月','二月'])
In:df
Out:
              一月  二月
product color
A       r       2   3
        g       4   5
B       r       6   7
        g       8   9

In: df.index
Out:
MultiIndex(levels=[['A', 'B'], ['g', 'r']],
           codes=[[0, 0, 1, 1], [1, 0, 1, 0]],
           names=['product', 'color'])
```

可以看出，df 的索引是两级索引，分别称为 0 级和 1 级索引，索引分别命名为 product 和 color。创建多级索引数据框还可以使用下面的等效命令。

```
In: df=DataFrame(np.arange(2, 10).reshape(4, 2),
index=[['A', 'A', 'B', 'B'],['r', 'g', 'r', 'g']], columns=['一月', '二月'])
In: df.loc['B']                      # 查看 B 类产品
Out:
   一月  二月
r   6   7
g   8   9
In: df.loc[('B', 'r')]               # 查看 B 类中的红色 r 产品
Out:
一月     6
二月     7
Name: (B, r), dtype: int32

In: df.loc[(slice(None), 'r'), :]    # 查看所有的红色 r 产品
Out:
     一月  二月
A r   2   3
```

```
B r   6   7
```

上例参数表中的 slice(None)是用于多级索引的切片语法，此处表示 0 级索引值任意，1 级索引值应为'r'。注意，这里不能写为(:, 'r') 。参数中最后的“:”代表所有列。多级索引的语法表达方式要复杂一些，一般多用于数据的分类统计。

```
In: df.loc['A'].sum().sum() , df.loc['B'].sum().sum()
Out: (14,30)           # 按 A/B 类求和
In: df.loc[(slice(None), 'r'), :].sum().sum()
Out: 18                # 求颜色 r 产品数量和
```

多级索引的数据框常用 stack()和 unstack()命令进行索引转换，以满足数据统计的要求。

```
In: df2 = df.unstack()      # 默认将最内层的 1 级行索引转为列索引
In: df2
Out:
  一月    二月
   g  r  g  r
A  4  2  5  3
B  8  6  9  7

In: df2.columns
Out:
MultiIndex(levels=[['一月', '二月'], ['g', 'r']],
           codes=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

转换得到的 df2 的列变为多级索引，使用类似 df2['一月']或 df2[('一月','g')]的语法访问。

```
In: df.stack()         # 将列索引（即列名）变为行索引
Out:                   # 行索引现有 3 级，第 0,1,2 三级
A  r  一月    2
      二月    3
   g  一月    4
      二月    5
B  r  一月    6
      二月    7
   g  一月    8
      二月    9
dtype: int32
```

8.5.8 字符串处理

字符串是最常见的数据类型。Pandas 为字符串提供了形如“obj.str.方法()”的一系列命令支持。为演示字符串的处理，下面构造一个由几部英文电影名构成的 Series。

```
In: s = Series(['Beauty and the Beast', 'Captain America: Civil War',
        'Jurassic World', 'Toy Story'])
In: s.str.len()        # 返回字符串长度
Out:
0    20
1    26
2    14
3     9
In: s.str.split()      # 分割字符串
```

```
Out:
0          [Beauty, and, the, Beast]
1    [Captain, America:, Civil, War]
2                  [Jurassic, World]
3                       [Toy, Story]
In: s.str[:6]          # 字符串切片
Out:
0    Beauty
1    Captai
2    Jurass
3    Toy St

In: s.str.contains('War')     # 测试电影名中是否包含 War
Out:
0    False
1    True
2    False
3    False
dtype: bool

In: s.str.replace(' ', '-')   # 字符替换，用横线-替换空格
Out:
0           Beauty-and-the-Beast
1     Captain-America:-Civil-War
2                 Jurassic-World
3                    Toy-Story
dtype: object
```

示例的字符串方法和 Python 自带的字符串函数功能类似，但更方便。这些方法一般在数据清洗、转换时使用。

8.6　分组统计

8.6.1　分组对象概述

Pandas 支持数据分组操作，其功能类似于数据库中的 group by（分组统计）。先构造测试数据如下。

```
In: df = pd.DataFrame({'color': ['red', 'white', 'black', 'red', 'black', 'red'], 'size':
['s','m','b','s','m','b'], 'price': [10, 11, 12, 20, 21, 22], 'quantity': [1, 2, 3, 3, 4, 5]})
In: df
Out:
   color   size   price   quantity
0    red    s      10         1
1  white    m      11         2
2  black    b      12         3
3    red    s      20         3
4  black    m      21         4
5    red    b      22        5
```

我们对这批数据按 color 或 size 分组，继而对每组进行相应的统计。

```
In: g = df.groupby('color')   # 按 color 分组
In: type(g)
Out: pandas.core.groupby.groupby.DataFrameGroupBy
```

分组得到的 g 变量是一个 DataFrameGroupBy 对象，它实际上还没有进行任何分组统计，仅含有一些有关分组如何划分的关键信息。分组对象具有很多特殊的属性和方法，如表 8.4 所示。

表 8.4 Pandas 分组对象的属性和方法

分组对象的属性和方法	描　　述
g.ngroups	分组数目属性
g.size()	列出每个分组所含的数据个数
g.sum(), g.mean(), g.std()	对每组求和、均值、标准差
g.groups	列出每个分组包含的数据索引编号
g.head(n), g.nth(n)	列出分组的前 n 个数据，第 n 个数据
g.describe()	对分组进行统计，返回一组常用统计量
g.agg([函数 1，函数 2])	对分组数据按指定函数进行统计

```
In: g.ngroups                          # ngroups 分组数
Out:3

In: g.groups                           # 列出每个分组包含的数据索引编号
Out:
{'black': Int64Index([2, 4], dtype='int64'),
 'red': Int64Index([0, 3, 5], dtype='int64'),
 'white': Int64Index([1], dtype='int64')}
```

如果用 for 循环遍历分组对象，那么遍历时每次取出的是一个组名和组内容，代码如下。

```
In: for  name,  group  in  g:
        print(name)                    # 输出组名
        print(group)                   # 组内容

black
   color size  price  quantity
2  black    b     12         3
4  black    m     21         4
red
  color size  price  quantity
0   red    s     10         1
3   red    s     20         3
5   red    b     22         5
white
   color size  price  quantity
1  white    m     11         2
```

要注意分组后不要修改原数据框，否则统计结果和修改后的新数据框会不一致。

8.6.2 分组对象的统计方法

分组对象支持的常用统计方法示例如下。

```
In: g.size()                    # 列出每个分组的数据个数
Out:
color
black    2
red      3
white    1
dtype: int64

In: g.sum()                     # sum 只对数值列求和，非数值列未显示
Out:
       price  quantity
color
black     33         7
red       52         9
white     11         2
```

分组对象可以使用 get_group()方法得到指定的分组数据。

```
In: g.get_group('black')        # 指定返回 black 组数据
Out:
   price  quantity size
2     12         3    b
4     21         4    m

In: g.head(2)                   # 取每个分组的头 2 个数据
   color size  price  quantity
0    red    s     10         1
1  white    m     11         2
2  black    b     12         3
3    red    s     20         3
4  black    m     21         4

In: g.nth(0)                    # 取每组的第 0 个数据
Out:
      size  price  quantity
color
black    b     12         3
red      s     10         1
white    m     11         2
```

分组对象的数值列可以执行 describe()操作，返回一组统计值。

```
In: g.price.describe()   # 对 price 列做 describe，得到一组常用统计量
Out:
       count   mean      std   min    25%   50%    75%   max
color
black    2.0  16.500000  6.363961  12.0  14.25  16.5  18.75  21.0
red      3.0  17.333333  6.429101  10.0  15.00  20.0  21.00  22.0
white    1.0  11.000000      NaN   11.0  11.00  11.0  11.00  11.0
```

分组对象有一个 aggregate 合计方法（可简写为 agg），它允许传递多个统计函数，因此可以一次性得到多个统计值，示例如下。

```
# 对 quantity 列求和、均值、最大值、最小值
In: g.quantity.agg((np.sum, np.mean, np.max, np.min))
Out:
       sum  mean  amax  amin
color
black    7   3.5     4     3
red      9   3.0     5     1
white    2   2.0     2     2

In: g.quantity.agg([('均值','mean'), ('最大值','max')])      # 定义列名
Out:
     均值  最大值
color
black  3.5    4
red    3.0    5
white  2.0    2
```

8.6.3 数据透视表

Excel 中提供一个很有用的数据透视表功能，Pandas 提供了类似的命令 pivot_table()，示例如下。

```
In: df                  # 沿用前面的示例数据 df
Out:
   color size  price  quantity
0    red    s     10         1
1  white    m     11         2
2  black    b     12         3
3    red    s     20         3
4  black    m     21         4
5    red    b     22         5

In: df.pivot_table(index='color', columns='size', values='quantity', aggfunc='sum')
Out:
size     b    m    s
color
black  3.0  4.0  NaN
red    5.0  NaN  4.0
white  NaN  2.0  NaN
```

参数表中 index 指定分组索引列，columns 指定列名，values 指定要计算的列，aggfunc 指定计算方法。例如，'sum'是求和，'mean'是求平均值，'count'是计数。

8.7 时间序列

我们日常处理的数据很多都与时间相关，如股票数据、穿戴式设备每日采集的身体健康数据。Pandas 最初研发的目的是作为金融数据分析包，它提供了丰富的时间序列处理方法，不同索引的时间序列之间的算术运算会自动按日期对齐。

8.7.1　Pandas 中的时间函数

1. to_datetime()

Pandas 提供 to_datetime()函数用于将字符串转换为时间值，该函数能识别不同格式的日期字符串。

```
In: pd.to_datetime('2019-2-20')                    # 年-月-日
Out: Timestamp('2019-02-20 00:00:00')
In: pd.to_datetime(['2/20/2019', '2019.2.20'])     # 不同格式的日期字符串都可转换
Out: DatetimeIndex(['2019-02-20', '2019-02-20'], dtype='datetime64[ns]', freq=None)

In: pd.to_datetime('2019-2-20 15:30:00')           # 年-月-日 时:分:秒
Out: Timestamp('2019-02-20 15:30:00')

In: today = pd.datetime.now()                      # 生成今天的日期
In: today + pd.DateOffset(days=3)                  # 后推 3 天
Out:Timestamp('2019-02-23 19:08:02.159070')
In: today + pd.DateOffset(years=1, months=6)       # 后推 1 年 6 个月
```

如下数据框索引类型初始不是日期类型，可用 to_datetime()方法将其转换为日期类型，以便于后续按日期段处理。

```
In: df=DataFrame(np.arange(2,5),index=['2019-1-1','2019-1-5','2019-1-10'], columns= ['A'])
In: df.index     # 初始索引的数据类型是 object，不是日期型
Out: Index(['2019-1-1', '2019-1-5', '2019-1-10'], dtype='object')

# 将索引的数据类型转为日期型
In: df.index = pd.to_datetime(df.index)
In: df.index
Out: DatetimeIndex(['2019-01-01','2019-01-05','2019-01-10'],
     dtype='datetime64[ns]', freq=None)
```

2. date_range()

date_range()用于产生指定日期段内的一系列日期值。调用格式为

```
pd.date_range(起始日期, 结束日期, periods=周期数, freq=日期频率)
```

表 8.5 中给出了 date_range()函数的常用 freq 参数表。

表 8.5　date_range()函数的常用 freq 参数表

参 数 名	含　义
H	时
T	分
S	秒
L	毫秒
D	日历日（即每天，此为默认值）
B	商业日（只含周一至周五，不含周六、日）
W	周
M	月底（如 1 月 31 日，2 月 28 日，6 月 30 日）
MS	月初（如 1 月 1 日，2 月 1 日，6 月 1 日）

（续表）

参 数 名	含 义
Q	季末（如 3 月 31 日，6 月 30 日）
QS	季初（如 1 月 1 日，4 月 1 日）
A	年底
AS	年初

```
In: pd.date_range('2019-02-01', '2019-02-28')                # 默认频率为 1 天
Out:                                                         # 共生成 28 个日期索引
DatetimeIndex(['2019-02-01', '2019-02-02', '2019-02-03', '2019-02-04',
                    …       …     …   …
               '2019-02-25', '2019-02-26', '2019-02-27', '2019-02-28'],
               dtype='datetime64[ns]', freq='D')

In: pd.date_range('2019-02-01', '2019-02-28', freq='3D')     # 频率为每 3 天
Out:
DatetimeIndex(['2019-02-01', '2019-02-04', '2019-02-07', '2019-02-10',
               '2019-02-13', '2019-02-16', '2019-02-19', '2019-02-22',
               '2019-02-25', '2019-02-28'],
               dtype='datetime64[ns]', freq='3D')

In: pd.date_range('2019-01-01', periods=3)                   # 生成 3 个数据
Out:DatetimeIndex(['2019-01-01', '2019-01-02', '2019-01-03'],
    dtype='datetime64[ns]', freq='D')

In: pd.date_range('2019-01-01', periods=6, freq='3D')   # 每 3 天 1 个日期，共 6 个数据
Out:DatetimeIndex(['2019-01-01', '2019-01-04', '2019-01-07', '2019-01-10',
               '2019-01-13', '2019-01-16'],
               dtype='datetime64[ns]', freq='3D')

In: pd.date_range('2018-01-01', '2018-06-30', freq='MS')     # 每月初
Out:DatetimeIndex(['2018-01-01', '2018-02-01', '2018-03-01', '2018-04-01',
               '2018-05-01', '2018-06-01'],
               dtype='datetime64[ns]', freq='MS')

In: pd.date_range('2018-10-01', '2018-10-02', freq='2H')     # 每 2 小时
Out:DatetimeIndex(['2018-10-01 00:00:00', '2018-10-01 02:00:00',
                   '2018-10-01 04:00:00', '2018-10-01 06:00:00',
                    …   …   …
                   '2018-10-02 00:00:00'],
                  dtype='datetime64[ns]', freq='2H')
```

有时需要对日期序列数据做 shift（移动）转换，以便计算相邻日期之间的数据变动。

```
In: np.random.seed(7)
In: dates = pd.date_range('2018-9-1', periods=4)        # 生成 4 个日期索引值
In: s = Series(np.random.rand(4), index=dates)
In:s
```

```
Out:
2018-09-01    0.076308
2018-09-02    0.779919
2018-09-03    0.438409
2018-09-04    0.723465
Freq: D, dtype: float64

In: s.shift(1)                          # 后移一个数据位
Out:
2018-09-01         NaN
2018-09-02    0.076308
2018-09-03    0.779919
2018-09-04    0.438409
Freq: D, dtype: float64

In: s.shift(-2)                         # 前移两个数据位
Out:
2018-09-01    0.438409
2018-09-02    0.723465
2018-09-03         NaN
2018-09-04         NaN
Freq: D, dtype: float64

In: s - s.shift(1)                      # 计算后一天相对前一天的变动值
Out:
2018-09-01         NaN
2018-09-02    0.703611                  # 即 2 日的数据 – 1 日的数据
2018-09-03   -0.341510                  # 即 3 日的数据 – 2 日的数据
2018-09-04    0.285056                  # 即 4 日的数据 – 3 日的数据
Freq: D, dtype: float64

# 计算变动百分比，map 方法将每个值按 format 指定的格式转换为百分比
In: ((s - s.shift(1)) / s.shift(1)).map(lambda x: format(x, '.2%'))
Out:
2018-09-01       nan%
2018-09-02    922.06%
2018-09-03    -43.79%
2018-09-04     65.02%
Freq: D, dtype: object
```

8.7.2　时间频率变换

用时间作为索引时，可以方便地按时间段查看数据。

```
In: np.random.seed(7)
In: dates = pd.date_range('2018-1-1', periods=365)
In: s = Series(np.random.randn(365), index=dates)  # 生成 2018 年全年的随机数据
In: s['2018-1']                             # 选取 2018 年 1 月的数据，输出数据略
In: s['2018-02':'2018-04']                  # 选取 2018 年 2 月至 4 月的数据，输出数据略
```

对时间序列数据可用 resample()方法按不同频率进行重采样，然后对样本进行计算。

```
# 按 1 个月的频率重采样
In: s.resample("1M").mean()              # 按月求均值
Out:
2018-01-31   -0.094655
2018-02-28   -0.025065
2018-03-31    0.068379
…             …
In: s.resample("10D").sum()              # 每 10 天求和
Out:
2018-01-01    0.740609
2018-01-11   -2.822628
2018-01-21   -0.109718

# 每 10 天一次采样，返回每组样本的最大值、最小值
In: s.resample("10D").agg([np.max, np.min])
Out:
             amax      amin
2018-01-01  1.690526 -1.754724
2018-01-11  0.554580 -1.526525
2018-01-21  2.029072 -2.288315
```

上例返回结果的第一行表示 2018-01-01 至 2018-01-10 这 10 天中的最大值、最小值。

8.8 实例应用

前面介绍了 Pandas 的基本操作，下面来看三个实例应用。

8.8.1 泰坦尼克号数据集分析

泰坦尼克号是史上著名的邮轮，首航时共搭载 2224 人，但海难发生后仅 710 人生还。研究人员收集整理了乘客数据集，用于研究生还者与哪些因素有关。在机器学习算法示例中经常出现此数据集。本节使用 Seaborn 库中包含的 titanic 数据集进行一些探索。Seaborn 是一个图形库，Anaconda 平台已安装此模块。模块中包含了若干示例数据，详见 https://github.com/mwaskom/ seaborn-data。第 9 章中将使用其中的 iris 数据绘制散点图。

```
In: import  seaborn  as  sns
In: tit = sns.load_dataset('titanic')   # 读取数据集
```

读取后得到一个 DataFrame，查看数据量如下。

```
In: tit.shape
Out: (891, 15)          # 891 行 x15 列
In: tit.head(3)         # 显示前 3 行数据
Out:
   survived  pclass     sex   age  ...  deck  embark_town  alive  alone
0         0       3    male  22.0  ...   NaN  Southampton   no   False
1         1       1  female  38.0  ...     C    Cherbourg   yes  False
2         1       3  female  26.0  ...   NaN  Southampton   yes  True
[3 rows x 15 columns]
In: tit.columns         # 显示列名
```

```
Out:
Index(['survived', 'pclass', 'sex', 'age', 'sibsp', 'parch', 'fare',
       'embarked', 'class', 'who', 'adult_male', 'deck', 'embark_town',
       'alive', 'alone'], dtype='object')
In: tit.to_excel('titanic.xlsx', index=False)  # 保存为电子表格，False 表示不保存索引列
```

数据集的列数较多，因此不便在 IPython 环境下查看。可以将数据保存为电子表格，然后在 Excel 中浏览数据全貌。数据集的结构如表 8.6 所示。表中的一些标注是为了让读者快速了解数据集，后面的示例中将演示如何得到这些统计数据。

表 8.6　泰坦尼克号幸存者数据集的字段说明

英文字段名	中文含义	数据值描述
survived	是否生还（数字 0,1）	0（no），1（yes）
pclass	客舱等级（数字 1,2,3）	1,2,3 对应一、二、三等舱
sex	性别	male，female
age	年龄	浮点，177 条记录此字段缺失
sibsp	船上兄弟姐妹的人数	整数，283 条记录此字段值>0
parch	船上父母和孩子的人数	整数，213 条记录此字段值>0
fare	客运票价	浮点
embarked	登船港口（单个字符）	三类：S，C，Q
class	客舱等级（字符描述）	三类：First，Second，Third
who	类型	三类：man，woman，child
adult_male	是否成年男性	True，False
deck	舱面	A C D E G，688 条记录此字段缺失
embarked_town	登船港口全名	共三个港口
alive	是否生还	字符：no，yes
alone	是否单独一人	True（537 条）、False（354 条）

survived 字段只有 0 和 1 两种取值，1 代表生还。计算平均生还率的程序如下。

```
In: tit['survived'].mean()          # 平均生还率
Out: 0.3838383838383838
In: tit.isnull().sum()              # 查看各列数据的缺失情况
Out:
survived         0
pclass           0
sex              0
age            177
sibsp            0
embarked         2
deck           688
...  ...                            # 省略
dtype: int64
```

上面的数据显示，age 列有 177 个 NaN 值（缺失数据），deck 列有 688 个 NaN 值。下面查看各列的取值情况：

```
In: tit.survived.unique()           # 取唯一值
Out: array([0, 1], dtype=int64)
In: tit.pclass.unique()             # 客舱等级
```

```
Out: array([3, 1, 2], dtype=int64)        # 分为 1,2,3 等级
In:tit.sex.unique()                       # 性别
Out: array(['male', 'female'], dtype=object)

In: tit.pclass.value_counts()             # 统计每类客舱记录数
Out:
3    491
1    216
2    184

In: tit.groupby('pclass')['survived'].mean()   # 按客舱等级统计生还率
Out:
pclass
1    0.629630
2    0.472826
3    0.242363
```

从统计数据可以看出，一等客舱生还率最高，三等客舱生还率最低，与之对应的是各等级客舱的票价也有很大差异。

```
# 统计每个等级的平均票价、最高价、最低价
In: tit.groupby('pclass').fare.agg(['mean', 'max', 'min'])
Out:
             mean     max    min
pclass
1       84.154687  512.3292  0.0
2       20.662183   73.5000  0.0
3       13.675550   69.5500  0.0

In: tit.groupby(['sex'])['survived'].agg(['sum','count','mean']) # 按性别统计生还率
Out:
        sum  count      mean
sex
female  233    314  0.742038
male    109    577  0.188908
```

由按性别统计的结果可见，女性生还率明显高于男性。上例中的 agg 函数功能强大，可以指定多个统计函数，本例同时完成了求和、计数、求平均值的统计。

表中的很多字段都可做类似的分类统计。我们将程序改写如下，以便输出各个字段的分类统计结果。

```
tit = pd.read_excel('titanic.xlsx')                # 读取 Excel 数据
fields = ['pclass', 'sex', 'who', 'embarked']      # 需做分类统计的字段
for  field  in  fields:                            # 按不同字段统计生还率
    print(tit.groupby(field)['survived'].agg(['sum', 'count', 'mean']))
for  field  in  ['sex', 'who', 'pclass', 'alone']:  # 按不同字段计数
    print('\n', field)
    print(tit.transform(field).value_counts())
```

上例中的 transform 是 Pandas 中的一个特殊函数，详情请查阅 Pandas 的帮助文档。此处将字符串形式的字段名转换为合法的 tit 表的列名，若只写 tit.field 则会报错。

表中的 parch 表示船上父母和孩子的人数，此字段大于 0 时表示家庭记录。

```
In: tit[tit.parch > 0]['survived'].mean()
Out: 0.5117370892018779
```

约 0.51 的生还率明显高于平均值 0.38，表明确实是家庭优先登上救生艇。

```
In: tit[(tit.age < 16)].groupby(['pclass'])['survived'].agg('mean')  # 16 岁以下儿童
Out:
pclass
1    0.833333
2    1.000000
3    0.431034
```

统计 16 岁以下孩子的生还率，可以发现孩子的生还率很高，尤其是二等舱中生还率达到了惊人的 100%。

下面设一个年龄区间段，统计每个年龄段的人数。考虑到年龄有很多缺失值，我们选择 ffill 模式填充（即缺失数据用其上一条记录的值填充），填充方法不一定合理，此处仅为演示如何处理缺失值。

```
In: tit.age.fillna(method='ffill', inplace=True)        # 填充缺失值
In: ages=[0,6,16,40,60,100]                             # 年龄区间段
In: pd.cut(tit.age,bins=ages).value_counts()            # 各年龄段人数统计
Out:
(16, 40]     576
(40, 60]     162
(6, 16]       65
(0, 6]        64
(60, 100]     24
```

上面以 titanic 数据集为例进行了简单的数据统计分析。Pandas 还有很多非常好用的功能，期待大家进一步探索。

8.8.2　电影票房统计

Tushare 是一个中文财经数据接口，其官网为 http://tushare.org，第 10 章将对其进行重点介绍。本节利用该接口从网上下载国内电影票房数据，然后做一些基本的统计分析。这个接口必须先安装。进入 Windows 的命令行窗口，执行下面的命令安装 Tushare：

```
pip   install   tushare
```

安装 Tushare 时默认从国外网站下载文件，网站访问速度很慢或无法访问时，可增加一个-i 参数，以指定使用清华的镜像网站，如下所示：

```
pip  install  tushare  -i  https://pypi.tuna.tsinghua.edu.cn/simple
```

安装后，若在 IPython 中执行 import tushare 未报错，则表示安装成功。

```
In: import  tushare  as  ts                 # 引入 Tushare
In: df = ts.month_boxoffice('2019-07')      # 下载 2019-07 票房
In: df.columns
Out:
Index(['Irank', 'MovieName', 'WomIndex', 'avgboxoffice', 'avgshowcount',
       'box_pro', 'boxoffice', 'days', 'releaseTime'], dtype='object')
In: df.iloc[:2, :7]                         # 查看前 2 行:前 7 列数据
Out:
  Irank MovieName WomIndex avgboxoffice avgshowcount box_pro boxoffice
0     1   哪吒之魔童降世         37           35     22.7    130538
1     2  扫毒 2：天地对决         36           17     22.6    129897
```

ts.month_boxoffice()可以读取指定月的月票房数据并返回一个 DataFrame 对象。返回的数据框对象共有 9 列，含义分别为：Irank，排名；MovieName，电影名称；WomIndex，口碑指数（很多缺失值）；avgboxoffice，平均票价；avgshowcount，场均人次；box_pro，月度占比；boxoffice，单月票房（万元人民币）；days，月内上映天数；releaseTime，首映日期。返回数据共 11 行，前 10 行是当月排名前十的电影，第 11 行是其他所有电影的合计票房。

```
In: df.iloc[-2:, :7]                        # 查看后两行
Out:
   Irank MovieName WomIndex avgboxoffice avgshowcount box_pro boxoffice
9     10    跳舞吧！大象          32            6     0.7      3838
10   999        其他              32            6     6.2     35643
```

Tushare 可提供从 2008 年 1 月开始至今的月票房数据。下面的代码每次下载一个月的数据并将新数据添加到已有数据集中，最后将数据保存为 Excel 文件。

```
import  tushare  as  ts
movie = DataFrame()                              # 生成一个 DataFrame 对象
for  year  in  range(2008, 2020):                # 2008—2019 年
    for  mon  in  range(1,13):                   # 1～12 月
        month='{:4d}-{:02d}'.format(year, mon) # 2008-01 格式
        df=ts.month_boxoffice(month)             # 下载指定月票房
        df['month'] = month                      # 添加月份列，记录数据对应的月份
        movie = movie.append(df, ignore_index=True)  # 将 df 追加到 movie 中
movie.to_excel('moviebox.xlsx', index=False)   # False 表示不保存索引列
```

上面的 df['month']=month 行很重要，每条数据一定要记录下所属的月份以便进行后续分析。查看保存的 moviebox.xlsx 文件后，发现最后几行都是“其他”数据行，但数据值是空白的，这是本年度的剩余月份，还没有实际数据。下载的数据初始都是字符串，所以不能直接做算术运算。将数据保存为 Excel 文件后，再使用 pd.read_excel()读取时会自动转换为数值或 object 类型。下面对数据做一些简单的处理。

```
In: m = pd.read_excel('moviebox.xlsx')    # 读入数据
In: m[m.boxoffice.isnull()]               # 显示最后几行票房空白的“其他”数据
In: m = m[m.boxoffice.notnull()]          # 只保留票房非空数据（剔除末尾空白行）
# 原数据中不含人数，按“单月票房（万元人民币）/平均票价”计算人数（取整），添加新列 people
In: m['people'] = (m.boxoffice * 10000 / m.avgboxoffice).astype('int')
# 将数据再保存为 boxmonth.xlsx，后续用此文件
In: m.to_excel('boxmonth.xlsx', index=False)
```

原始数据有 9 列，我们增加了 month 月份列和 people 观影人数列。处理数据时要根据需要增删数据列。下面对 boxmonth.xlsx 文件中的数据进行统计分析。

```
In: movie = pd.read_excel('boxmonth.xlsx')
In: movie.loc[:, ['boxoffice', 'people']].sum()  # 总票房（单位万元人民币），总观影人数
Out:
boxoffice      34272885
people       9815556073

# 查看 2008—2019 年电影十大票房排行榜
In: m = movie[movie.MovieName!='其他']              # 先排除“其他”行
In: m.groupby('MovieName').boxoffice.sum().sort_values(ascending=False)[:10]
# 分组排序
```

```
Out:
MovieName
战狼 2                    566339.0
流浪地球                  464836.0
复仇者联盟 4：终局之战    423944.0
# ……
```

我们可以统计年度票房和月度票房，然后绘制对比图形。

```
# 由于 2019 年的月份数据不全，因此排除 2019 年的数据，只统计 2008—2018 年的数据
In: m = movie[movie.month.str[:4] != '2019']
# 按年度，str[:4]取出年份，以此分类统计，sort_index 按索引年度顺序排列
In: ybox = m.groupby(m.month.str[:4]).boxoffice.sum().sort_index()
In: ybox[::-1]
Out:
month
2018    6069736
2017    5583126
2016    4552666
# 作图（图形将在第 9 章介绍），如图 8.2 所示
In: ybox.plot(title='年票房', marker='o', fontsize=14)

# 按月份，str[5:7]取出月份，以此分类统计
In: mbox = m.groupby(m.month.str[5:7]).boxoffice.sum().sort_index()
In: mbox.plot(title='月票房', marker='o', fontsize=14)    # 如图 8.3 所示
```

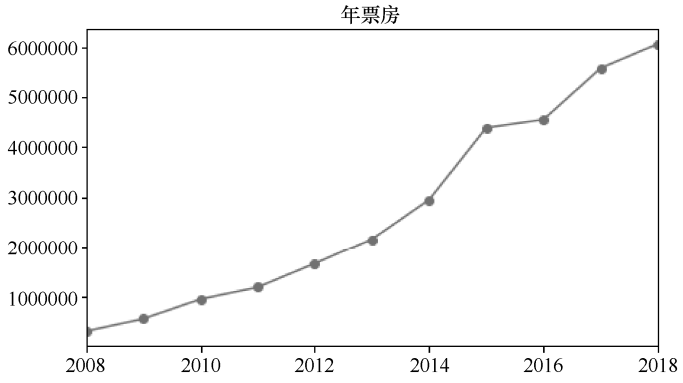


图 8.2　2008—2018 年电影年度票房对比图

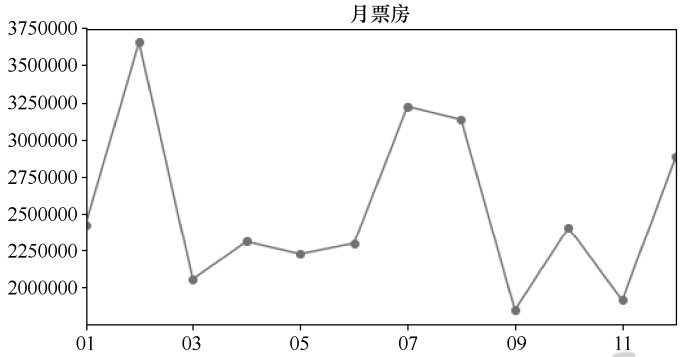


图 8.3　电影月度票房对比图

从图 8.2 和图 8.3 可以看出，近十年的年度票房增长很快，呈现井喷势头：从 2008 年的约 30 亿元人民币增加到了 2018 年的 600 亿元人民币。电影月度消费差异很大，春节和暑假消费爆棚，所以这两个档期也是电影公司必争的黄金档期。

```
# 按年度统计观影人数，ascending=False 降序
In: p = m.groupby(m.month.str[:4]).people.sum().sort_values(ascending=False)
In: p
Out:
2018    1718045678
2017    1622494840
# ... ...

# 计算年度人均票价“年度总票房（万元人民币）/观影人数”，保留 1 位小数
In: np.round(ybox*10000/p, 1)
Out:
2008    28.8
2009    30.8
2010    35.3
```

运算 ybox*10000/p 体现了 Pandas 索引运算的优势，两列统计数据都以年份为索引，在运算时自动按年份匹配。

8.8.3 股票基本面统计

Tushare 中有一个下载股票基本面数据的接口，可以一次性下载所有股票的基本数据，这对观察股票的整体市场情况很有帮助。

```
In: import  tushare  as  ts
In: stock = ts.get_stock_basics()              # 下载股票基本面数据
In: stock.to_excel('stock.xlsx')               # 保存为电子表格
In: stock.shape
Out: (3678, 22)                                # 3678 只股票
```

数据集规模为 3678×22，每行是一只股票的基本数据。字段详情请查看 Tushare 的网站。本节用到的数据列有：code，股票代码；name，名称；industry，所属行业；area，地区；pe，市盈率；totals，总股本（亿元人民币）；esp，每股收益；timeToMarket，上市日期。

下面从电子表格文件中读取数据，注意股票代码 code 列的处理细节。Pandas 读取数据时，总是试图将数据自动转换为数值类型。深市类似'002522'的股票代码读入后，将丢失前导字符“00”，变为整数 2522，因此读取时特意指定 code 字段为字符串。

```
In: df = pd.read_excel('stock.xlsx', dtype={'code': 'str'})  # code 字符串类型
In: df.set_index('code', inplace=True)             # 将 code 设为索引列
In: df.loc['002522']                               # 显示某只股票基本面数据
In: len(df.industry.unique())                      # 显示行业数
Out: 110
In: len(df.area.unique())                          # 显示地区数（即股票的归属省份）
Out: 32
In: df.groupby('area').area.count().sort_values(ascending=False)
Out:                                        # 按地区统计上市公司数量，体现地区经济实力
area
浙江     445
```

```
江苏    414
北京    332
广东    313
上海    297
深圳    290
# ……
```

由上面的统计结果可见，经济越发达、越有活力的地区，上市公司的数量越多。读者还可以按行业进行类似的统计。数据框中的 timeToMarket 字段代表上市日期，其数据是格式形如“20190315”的整数类型。我们可以提取出其中的年份以统计每年的股票发行数量。

```
year = df.timeToMarket.astype('str').str[:4]  # 转换为字符串，提取前 4 位的年份
yearnum = df.groupby(year).name.count()       # 按年份统计，得到每年股票发行量
# 数据集中有几只股票没有发行年份（年份为 0），作图时排除 0 年份
yearnum[yearnum.index!='0'].plot(fontsize=14, title='年 IPO 数量') #如图 8.4 所示
```

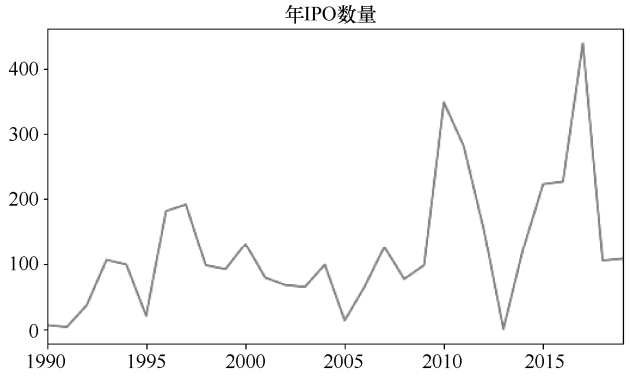


图 8.4　年 IPO 数量图

从图中可见 IPO 发行年份的几个高点和国内股票市场的几次牛市时间对应，熊市时发行数量跌入低谷。下面计算市场的平均市盈率 pe，这是衡量股票市场估值的重要参数。

```
In: df.pe.mean()                # 简单的算术平均 pe
Out: 77.06191408374107
```

观察数据集发现，数据集中亏损股票的 pe 为 0，因此考虑剔除亏损股票。

```
In: df[df.pe > 0].pe.mean()     # 剔除亏损股票后计算 pe 均值
Out: 90.84414102564092
```

上面的 pe 是简单的算术平均，以市值为权重的加权 pe 可能更能准确地反映市场状况。由于下载的数据集中没有总市值和股票单价，因此只能根据现有的字段推算总市值。由某些列的值计算新的列值也是数据处理中常有的情况。这里推算总市值的依据如下：

股票单价 ＝4*esp（每股收益）*pe（市盈率）

总市值 ＝ 股票单价*totals 总股本（亿元人民币）

数据集中每股收益 esp 是单个季度的收益，因此计算全年收益时要乘以 4。

```
In: df['tvalue'] = 4 * df.esp * df.pe * df.totals  # 计算总市值，增加新列 tvalue
In: np.sum(df.pe * df.tvalue) / df.tvalue.sum()    # 计算以市值为权重的加权 pe
Out: 48.35114474026812
```

上面的计算结果反映了某次季报后的市场加权 pe 情况，结果和市场真实值相比有差异。这是因为股票的每个季度收益不同，因而不能简单按“4*单季收益”来计算全年收益。

我国股票市场现分为沪市（股票代码 60 开头）、深圳主板（股票代码 00 开头）、创业板（股

票代码 30 开头）及最新上市的科创板（股票代码 68 开头）。如下代码可计算不同板块的 pe 值和股票数。

```
In: df['board'] = df.index.str[:2]     # 取 code 的前 2 个字符，新增 board 列
# 按板块类型统计 pe 均值，计数
In: df.groupby('board').pe.agg([('pe 均值', 'mean'), ('股票数', 'count')])
Out:
          pe 均值    股票数
board
00       63.644039   1399            # 深市
30      116.040832    769            # 创业板
60       67.464811   1482            # 沪市
68      184.909286     28            # 科创板
```

8.9 本章小结

本章介绍了 Python 生态圈重要的数据分析包 Pandas。与 NumPy 相比，Pandas 增加了自定义标签支持，既可用整数索引访问，又可用标签访问。单独的一行或一列是 Series，整个二维表是 DataFrame，创建时可以将列表、字典作为参数传递。

df['列名']或 df[['列 1','列 2']]访问列，df.iloc[整数索引]或 df.loc['标签']访问行，df.at[行,列]访问单个数据。

pd.read_csv()和 df.to_csv()存取 CSV 文件，pd.read_excel()和 df.to_excel()存取 Excel 文件。读/写中文文件时注意 encoding 字符编码参数。

Pandas 运算时默认忽略 nan 值，标签自动对齐运算。NumPy 可使用的通用函数，Pandas 都能使用。df.isnull()、df.notnull()用于 nan 值判断。reindex()用于索引整理，rename()用于重命名，duplicated()用于重复值判断，sort_index()用于按索引排序，sort_values()用于按数据值排序。

pd.merge()实现数据框连接，pd.cut()实现数据分段。df.groupby()实现数据分组，pivot_table()产生分组透视表。

Pandas 在时间序列处理上很有优势。date_range()产生时间序列索引，resample()按时间频率重采样。

习题

1. Pandas 最重要的两种基本数据结构是什么？试各用三种不同的方式创建这两类数据对象。
2. Series 和 NumPy 数组的区别是什么？
3. 对于 nan 值，Pandas 提供了哪些处理函数？
4. 举例说明 reindex()、set_index()和 reset_index()方法的作用。
5. 试举一例，演示数据框分组的各种操作。
6. Pandas 在时间序列上提供哪些函数？
7. Pandas 如何读/写 CSV 文件和 Excel 文件？
8. 构建两个具有相同字段名的数据框，用 pd.merge()方法完成连接。
9. 构建含有 1000 个随机整数的 Series，整数值在区间[1, 100]内，输出显示不重复的数据值，统计每个数据的出现次数。
10. 构建数据框，练习使用 pivot_table()分组透视表。

第 9 章　Matplotlib 绘图库

数据可视化是指以图形的方式展示数据。好的图表自己会“说话”，正所谓“一图抵千言”。人们在企业管理和科学研究中积累了大量的数据，这些数据在规模和维度上日益增长。为了揭示数据背后的深刻含义，我们需要对海量数据进行抽取、加工和提炼。前面介绍的 NumPy 和 Pandas 就是数据处理的好帮手。本章介绍的 Matplotlib 是 Python 平台下的绘图库，它为数据可视化提供了理想的解决方案。

9.1　Matplotlib 简介

Matplotlib 本是由 John D. Hunter 发起的一个开源项目，现在已逐步发展为 Python 生态圈中应用非常广泛的绘图库，它支持二维绘图和部分三维绘图。因为底层依赖于 NumPy，所以必须安装 NumPy 才能使用 Matplotlib。Matplotlib 借鉴了 MATLAB 的设计理念，熟悉 MATLAB 的读者可以很快上手。对于新手，Matplotlib 提供了非常丰富的文档和实例，方便用户学习该软件的各项功能与操作方法。

Matplotlib 的官方示例网站是 https://matplotlib.org/gallery.html。读者可以在这个网站看到很多精彩的示例，体验其强大的绘图和数据展示能力。

Matplotlib 提供交互绘图方式，用户可在交互环境中执行命令来实时地绘制与修改图形。生成的图像可以保存为多种格式，如 png、pdf、jpg、svg 等。本章依旧在 Anaconda 平台的 IPython 环境下学习 Matplotlib。

一般使用下面的命令导入 Matplotlib 图形库，本章后续都以 `plt` 指代该模块。

```
import  matplotlib.pyplot  as  plt          # 导入 pyplot 模块并命名为 plt
```

图形输出分为嵌入模式和独立窗口模式两种，在交互窗口中可用下面的两条命令控制图形的输出形式。嵌入模式在 IPython 的交互窗口中显示图形，图形显示后不能再修改。独立窗口模式在弹出的一个窗口中显示图形，图形可以放大、缩小和修改。

```
%matplotlib  inline                         # 设置嵌入模式显示图形
%matplotlib                                 # 设置独立窗口模式显示图形
```

以%开头的命令是 IPython 的所谓“魔法命令”，它是 IPython 特有的配置命令，标准 Python 不支持这类配置命令。这些魔法命令只能在 IPython 交互窗口中执行，而不能包含在 Python 源程序中。

9.1.1　绘图示例

在 IPython 交互窗口中执行下面的魔法命令将图形输出设为独立窗口模式。

```
%matplotlib                                 # 设置独立窗口模式
```

如下代码绘制一条简单曲线 $y = x^2$，如图 9.1 所示。软件绘图时仍然基于描点连线的原理，因此需要提供曲线上若干点的 x 坐标和 y 坐标，这些坐标可用 NumPy 的数组表达。

```
import  numpy  as  np
import  matplotlib.pyplot as plt            # 导入 plt
x = np.linspace(-2, 2, 50)                  # 在区间[-2, 2]内等间距产生 50 个点
y = x**2                                    # 计算平方值
```

```
plt.plot(x,y,color='blue',ls='--',label='y=x^2') # 绘制曲线并设定蓝色、破折线、图例
plt.xlabel('x', fontsize=14)              # x 轴标记字符 x
plt.ylabel('y', fontsize=14)              # y 轴标记字符 y
plt.title('Example', fontsize=18)         # 设置标题
plt.legend()                              # 显示图例（y=x^2）
plt.show()                                # 显示图形
```

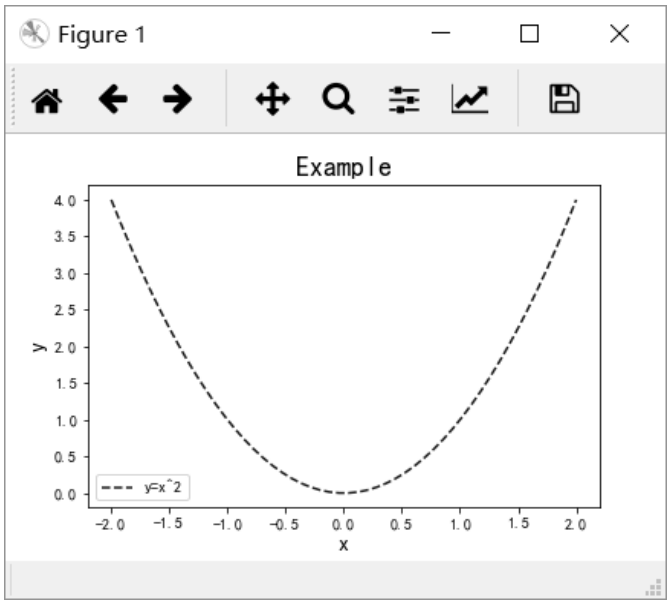


图 9.1　$y=x^2$ 曲线图（独立窗口模式）

图 9.1 是以独立窗口模式显示图形的。程序中调用的命令形式皆为“plt.方法()”。每个方法中可指定不同的参数，用于设定线条颜色、线条形状、图例文字等格式信息。独立图形窗口可放大、缩小，并可将图形保存为多种图片格式，支持后续的交互式修改。

图形窗口顶部有 8 个按钮，具体如下：

- 重置视图：显示最初的视图
- 上一个视图
- 下一个视图
- 移动工具：用于拖动绘图的显示区域
- 缩放工具：可以拖曳出一个矩形窗口以放大选定的图形区域
- 配置子图
- 设置图形参数
- 保存图片

下面再以嵌入模式绘制一条 $y=2x$ 的直线。要设置嵌入模式，需要在 IPython 交互窗口执行命令“%matplotlib inline”。

```
y = 2*np.arange(10)
plt.plot(y, color='b', ls='-', linewidth=2)    # 本例只提供了 y 值。蓝色、实线、线宽 2
plt.title('y=2*x', fontsize=18)
plt.show()
```

绘制二维图时一般应提供 x 轴和 y 轴两组数据。若只提供一个序列的数据，则 Matplotlib 会

假定提供的是 y 轴数据，并从 0 开始自动生成 x 轴上的顺序整数值，所以 x 轴的数据值会是[0,1,2,…,len(y)-1]。在嵌入模式下，图形显示在 IPython 的交互窗口中，嵌入模式下的图形只能查看，不能放大和缩小，也不能再进行交互式修改，如图 9.2 所示。

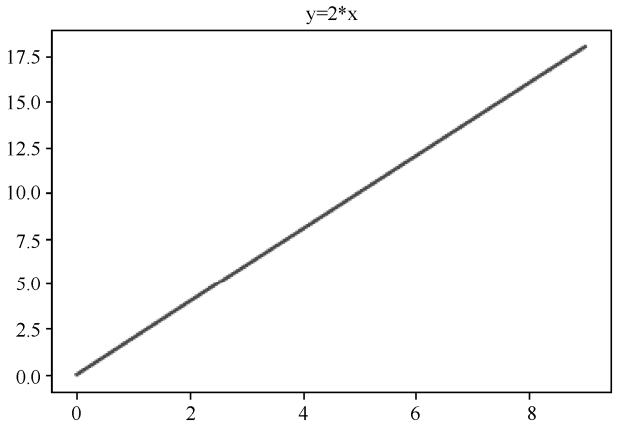


图 9.2　$y=2x$ 直线图（嵌入模式）

Matplotlib 支持两种绘图模式：一种是类 MATLAB 风格，主要通过命令“plt.方法()”绘图；另一种是面向对象绘图风格，采用这种风格的等效代码如下。

```
y = 2*np.arange(10)
fig = plt.figure()                          # 先生成一个 figure 对象，代表整个图形窗口
ax = fig.add_axes([0.1, 0.1, 0.6, 0.6])     # 再在 fig 上生成一个 axes，代表作图区域
ax.plot(y)                                  # 然后在 axes 上绘图
ax.set_title('y=2*x', fontsize=18)          # 设置标题
```

本章主要采用类 MATLAB 风格，目的是让初学者更容易掌握基本的绘图方法。

9.1.2　颜色、线型和标记符号

作图时通常需要指定线条的颜色、线型、标记符号等参数，语句格式如下：

```
plt.plot(x, y, ls='线型', color='颜色', marker='标记')
```

常用的各种格式符号如表 9.1、表 9.2、表 9.3 所示。

表 9.1　linestyle(ls)线型参数表

linestyle 参数表	描　述	linestyle 参数表	描　述
-	实线	:	虚线
--	破折线	None	空白（不画线）
-.	点画线		

表 9.2　常用 color(c)参数表

颜色名称	简　写	描　述	颜色名称	简　写	描　述
red	r	红色	yellow	y	黄色
black	k	黑色	white	w	白色
blue	b	蓝色	green	g	绿色
cyan	c	青色			

表 9.3 marker 标记符号表

标 记	描 述	标 记	描 述
o	圆圈	S（小写）	正方形
.	点	*	星号
D	菱形	d	小菱形
+	加号	x	X 号
v（小写）	一角朝下的三角形	<	一角朝左的三角形
>	一角朝右的三角形	^	一角朝上的三角形
None	空白		

标记符号有多种，读者可以用“plt.plot?”命令查看相关的文档。

```
x = np.arange(10)
y = x**2 + 2 * x + 5
plt.plot(x,y,c='r',ls=':',marker='o',markersize=10)   # 设置颜色、线型、标记符及标记大小
plt.plot(x, y, 'ro:')                                 # 简写，效果同上条命令
```

上面两条 plt.plot()命令设置了红色、虚线、圆圈标记符号，如图 9.3 所示。第二条画图命令采用了简写方式，将颜色、线型、标记缩写为一个字符串'ro:'。作折线图时如不设置格式符号，则默认的格式字符是'b-'，即蓝色实线。

可以一次性传送多组数据和格式符，以便在一幅图上绘出多条线，如图 9.4 所示。

```
x = np.arange(10)
y1 = x
y2 = x**2 + 3 * x + 4
y3 = 5 * x + 10
plt.plot(x, y1,'rs-', x, y2, 'b--', x, y3, 'go')   # 提供三组数据绘制三条线，见图 9.4
```

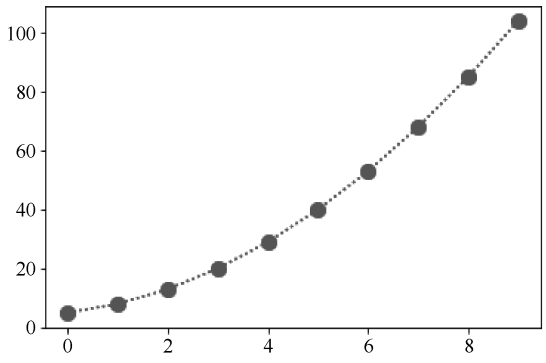


图 9.3 “ro:”红色虚线图

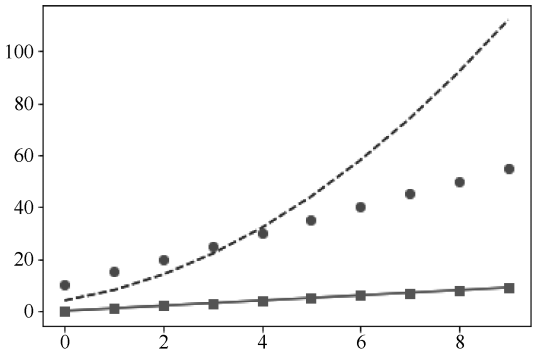


图 9.4 一图绘制多条曲线

9.1.3 plt 常用命令

Matplotlib 的绘图命令的一般形式为“plt.方法(参数)”，如表 9.4 所示。下面解释其中的一些常用命令项。

表 9.4　plt 的常用方法表

方法示例	解　　释	方法示例	解　　释
plt.plot()	折线图	plt.grid(True/False)	显示/不显示网格
plt.xlim(0,5)	设置 x 轴范围	plt.ylim(0,8)	设置 y 轴范围
plt.axis('equal')	设置 x/y 轴单位长度相等	plt.axis('on/off')	显示/不显示坐标轴
plt.xlabel('x 轴')	设置 x 轴标记文字	plt.ylabel('y 轴')	设置 y 轴标记文字
plt.title('图标题')	设置图形标题	plt.legend()	显示图例
plt.text(x,y,'text')	在指定坐标处显示文字	plt.savefig('a.png')	保存图片
plt.figure(figsize=(m,n))	设置图形大小	plt.style.use('风格')	设置绘图风格

plt.title()用于设置图形的标题。如果想限制坐标轴的显示范围，那么可用 plt.xlim()和 plt.ylim()函数分别设置 x 轴和 y 轴的坐标显示区域。

```
plt.plot(np.arange(10))
plt.title('Graph Title', fontsize=18)     # 设置标题
plt.xlim(1, 5)                            # x 轴显示范围[1,5]
plt.ylim(2, 6)                            # y 轴显示范围[2,6]
```

由于屏幕的横向分辨率和纵向分辨率不同，单位长度对应的横、纵屏幕点数也不同，所以在屏幕上显示一个圆时看起来像是椭圆。这时，可用 plt.axis('equal')将横、纵轴的屏幕单位长度设为相同。下面的语句是画一个半径为 3 的圆，其对比效果如图 9.5 和图 9.6 所示。

```
x = np.linspace(-3, 3, 100) # 在区间[-3, 3]内生成 100 个 x 值
y = np.sqrt(9 - x**2)  # 计算半径为 3 的圆上的点的 y 坐标
plt.plot(x, y, x, -y)  # 画半径为 3 的圆。2 组数据，先画上半圆，再画下半圆，见图 9.5
#plt.axis('equal')     # 设置 x/y 轴的单位长度相等。如删除本行注释符号，效果见图 9.6
```

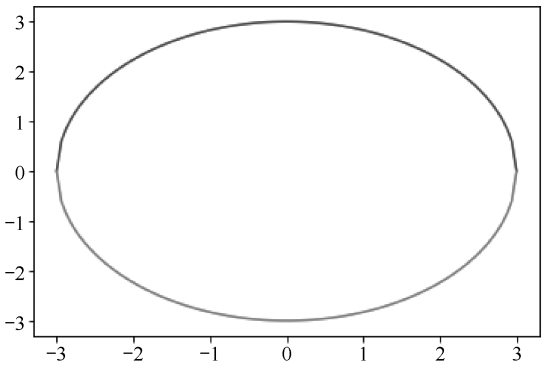


图 9.5　未设置 axis('equal')的圆

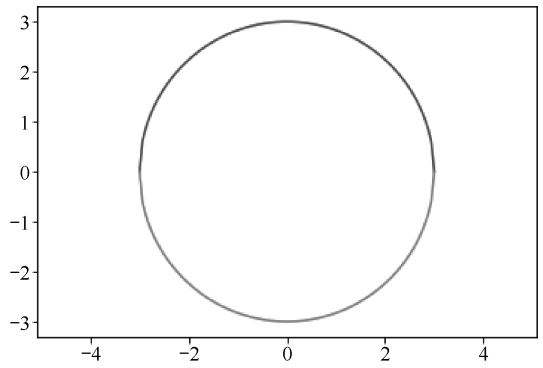


图 9.6　设置了 axis('equal')的圆

如果要突出图形而淡化其他内容，那么可用 plt.axis('off')隐藏坐标轴。如果想更清楚地观察数据值，那么可以用 plt.grid(True)显示网格。

```
x = np.linspace(-3*np.pi, 3*np.pi, 100)   # 在区间[-3π,3π]内生成 100 个 x 值
y = np.sin(x)                             # 计算 sin 值
plt.plot(x, y)                            # 画正弦曲线图
plt.grid(True)                            # 显示网格线，见图 9.7
plt.grid(axis='x')                        # 只显示垂直网格线
plt.grid(axis='y')                        # 只显示水平网格线
```

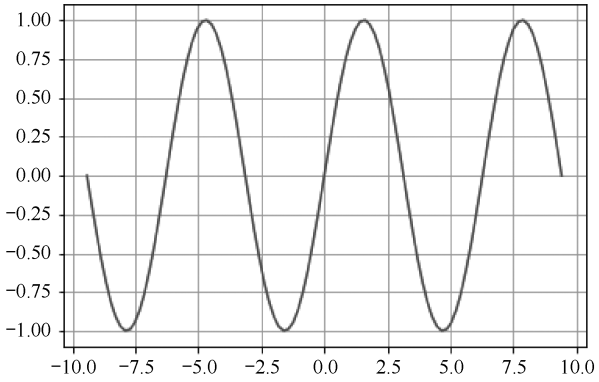


图 9.7　设置了网格线的图形

一幅图中经常包含多个子图形，可用参数 label='图例名'给各个子图加上图例说明，并用 plt.legend()命令显示图例。

```
x = np.arange(10)
y1 = 3*x
y2 = 5*x
plt.plot(x, y1, label='Line1')            # 线 1
plt.plot(x, y2, label='Line2')            # 线 2
plt.legend()                              # 按默认方式显示图例，见图 9.8
# loc=8 图例显示在下方并居中，frameon=False 图例不显示外边框，ncol=2 分 2 列
#plt.legend(loc=8, frameon=False, ncol=2) # 见图 9.9
```

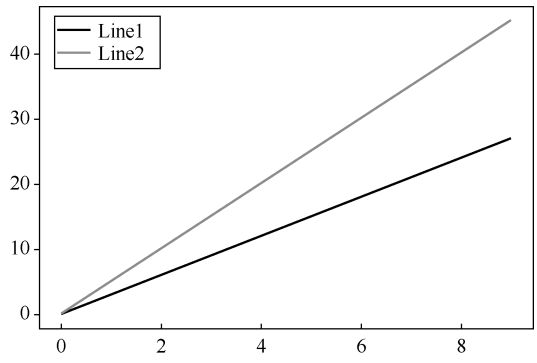


图 9.8　图例在默认位置（loc='best'）

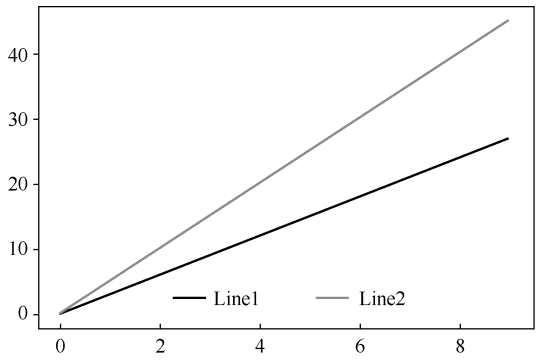


图 9.9　图例在指定位置（loc=8）

plt.legend()用于显示图例，可用参数 loc='位置'来规定图例的位置，参数值如表 9.5 所示。参数值用英文字符串或整数表达均可。plt.legend(loc='lower right')和 plt.legend(loc=4)的效果相同，都将图例显示在右下角。

表 9.5　plt.legend(loc='位置')参数表

位置字符串	整　数　值	位　置	位置字符串	整　数　值	位　置
best	0（默认值，自适应）	最佳	upper right	1	右上
upper left	2	左上	lower left	3	左下
lower right	4	右下	right	5	右
center left	6	左中	center right	7	右中

（续表）

位置字符串	整　数　值	位　置	位置字符串	整　数　值	位　置
lower center	8	中下	upper center	9	中上
center	10	居中			

plt.figure(figsize=(6,6))可设置图形的大小，单位为英寸。

plt.savefig('fig.png', bbox_inches='tight')可保存图片，bbox_inches='tight'表示保存时剪除图形周围的空白部分。可用不同的扩展名 png、jpg、svg 将绘制的图形保存为不同格式的图片。图片默认保存在当前工作目录中。

plt.style.use('风格')用于设置绘图风格，不同风格的图形显示视觉效果有差异。

```
In: plt.style.available              # 显示可用的风格名称
In: plt.style.use('ggplot')          # 设置为 ggplot 绘图风格
In: plt.style.use('seaborn')         # 设置为 seaborn 绘图风格
```

9.1.4　中文显示问题

Matplotlib 的默认配置文件无法正确显示中文，如果在图形中设置了中文标题名，那么中文将错误地显示为小方格。要正确显示中文，可用下面的两种解决方法。

方法 1：每次在程序头部设定中文字体，示例代码如下。

```
import  matplotlib.pyplot  as  plt
plt.rcParams['font.sans-serif'] = ['SimHei']   # 指定中文黑体字体
# 下面的 False 修正坐标轴上负号'-'显示为方块的问题
plt.rcParams['axes.unicode_minus'] = False
plt.xlim([-5, 5])                              # 测试：坐标轴上负号显示正常
plt.title('中文标题')                          # 测试：中文标题显示正常
```

该方法要求在每个程序中都指定中文字体，较烦琐，不太方便。

方法 2：修改 Matplotlib 配置文件以显示中文（推荐采用此方法）。此方法可在本机上一劳永逸地解决中文的显示问题，步骤如下。

1．定位 matplotlibrc 配置文件的位置

```
import   matplotlib                  # 导入库
matplotlib.matplotlib_fname()    # 显示 matplotlib 配置文件名
```

上面命令的显示内容随 Anaconda 版本和程序的安装位置变化。例如，在笔者机器上显示为 C:\\ProgramData\\anaconda35\\lib\\site-packages\\matplotlib\\mpl-data\\matplotlibrc。

打开文件夹 C:\ProgramData\anaconda35\Lib\site-packages\matplotlib\mpl-data，定位到配置文件 matplotlibrc，如图 9.10 所示。

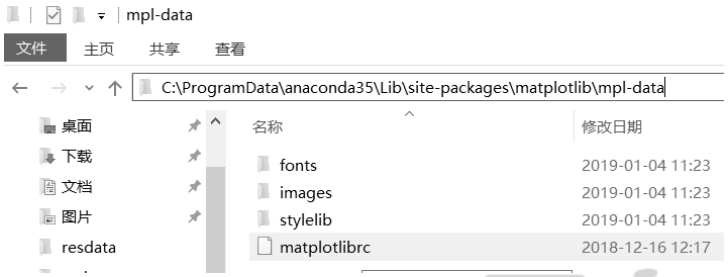


图 9.10　定位 matplotlibrc 配置文件

2．用记事本程序打开 matplotlibrc 配置文件

（1）查找到下面这一行：

```
#font.sans-serif : Bitstream Vera Serif, New Century Schoolbook, Century
```

将前面的注释符号#去掉，并在字体列表前添加 SimHei（黑体常规字体），即修改为

```
font.sans-serif : SimHei, Bitstream Vera Serif, New Century Schoolbook, Century
```

（2）设置坐标轴的负号正常显示，在配置文件中找到下面这一行：

```
#axes.unicode_minus  : True
```

去掉注释符号#，同时将 True 改为 False，即修改为

```
axes.unicode_minus  : False
```

保存配置文件，重启 Spyder 后新配置生效，图形中的中文输出正常。本章后续图形中的中文输出默认已按方法 2 处理，所以代码中不再设定中文字体。如果代码要在其他机器上运行，为确保中文显示正常，应采用方法 1 的写法。

9.2 几种常见的图形

Matplotlib 支持多种图形的绘制，如表 9.6 所示。前一节演示了折线图的画法，本节演示其他几种图形的画法。

表 9.6 Matplotlib 支持的常用图形类型

命 令 名	图形类型	命 令 名	图形类型
plt.plot()	折线图	plt.bar()	柱形图
plt.pie()	饼图	plt.barh()	水平条形图
plt.hist()	直方图	plt.scatter()	散点图
plt.boxplot()	箱线图		

表 9.6 中列出的图形和 Excel 中的统计图类似，但 Matplotlib 图形是用编程方式实现的，所以适合处理较大的数据量，且定制性更好。

9.2.1 柱形图

柱形图用于对一组数据值进行比较，是最常见的对比图形。纵置时称为柱形图，横置时称为条形图。

```
sign = list('abcde')
data = [0.2, 0.3, 0.4, 0.5, 0.1]
plt.bar(x=sign, height=data)     # 可简写为 plt.bar(sign, data)
plt.ylim([0, 0.7])               # 设置 y 轴的显示区域
for  x, y in enumerate(data):    # 将数据值标注在柱形上方
    plt.text(x, y+0.03, '{}'.format(y), ha='center', fontsize=14)
                                 # y+0.03 表示比 y 值高 0.03 的位置
plt.show()                       # 见图 9.11
```

如果绘图时采用的是独立窗口模式，则在 IPython 交互窗口中执行下面一行命令就可修改已显示图形的 x 轴上的标记文字，如图 9.12 所示。

```
In: plt.xticks(sign, ['s1','s2','s3','s4','s5'], fontsize=14)  # 设置 x 轴标记文字
```

绘制柱形图时至少要提供参数 x 和 height 两组数据。如果 x 是非数值型的，如字符数据，

那么 plt 依次顺序排列柱形；如果 x 是数值型的，那么 plt 在 x 轴的对应位置上显示柱形。柱形图还有很多参数，详情可用命令“plt.bar?”查看帮助。下面的示例演示了一些参数的效果。

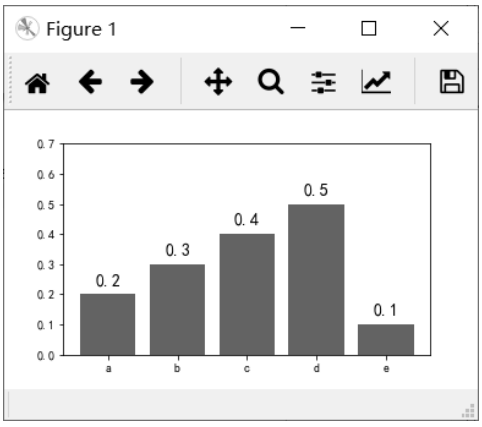


图 9.11　柱形图

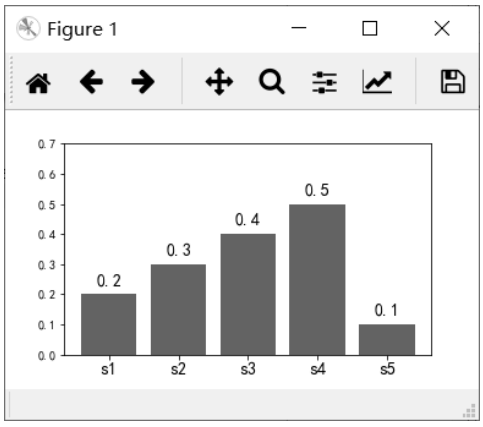


图 9.12　交互修改柱形图 x 轴标记文字

```
position = np.arange(1, 4)
data1 = [0.2, 0.3, 0.4]
w = 0.3                                          # 宽度值
plt.bar(position, data1,
        width=w, color='grey',                   # 宽度，内部填充色
        edgecolor='r', hatch='/',                # 边缘颜色，内部填充图案/
        tick_label=['一月', '二月', '三月'],      # 坐标轴标记
        label='A')                               # 图例
data2 = [0.4, 0.2, 0.4]
plt.bar(position, data2, width=w,
        bottom=data1,                            # data2 以 data1 为底部，产生堆积效果
        edgecolor='w', hatch='x', label='B')
plt.legend()                                     # 显示图例，见图 9.13
```

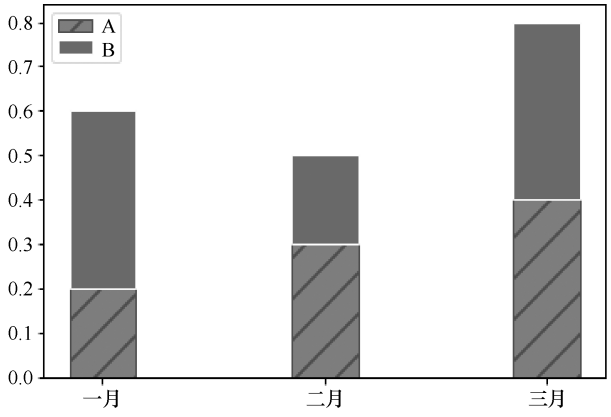


图 9.13　堆积柱形图（两组数据堆叠显示）

注意第二条绘图命令中的参数 bottom=data1，该参数表示第二组柱形以第一组的数据为底，这样就构成了堆积柱形图。

```
position = np.arange(1, 4)
```

```
data1 = [0.2, 0.3, 0.4]
data2 = [0.4, 0.2, 0.4]
w=0.3
plt.bar(position, data1, width=w)
plt.bar(position + w, data2, width=w)     # 调整 position 位置，簇状图，见图 9.14
```

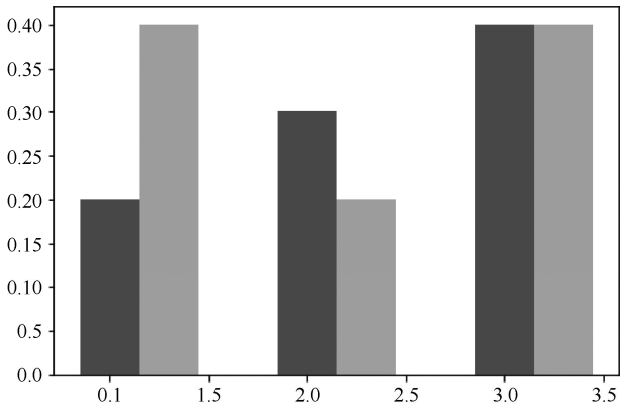


图 9.14　簇状柱形图

注意上例第二条绘图命令中的参数 position+w。加上 w 是为了调整柱形的位置，这样第二组柱形就紧邻在第一组的右侧，形成簇状柱形图。

除了绘制竖直柱形图，绘图库还提供水平条形图命令 plt.barh()。下面对两组数据做水平条形图对比。

```
data1 = np.array([5, 10, 33, 21])
data2 = np.array([5, 15, 28, 18])
x = np.arange(4)
plt.barh(x, data1, hatch='x')         # 图 9.15 右侧
plt.barh(x, -data2, hatch='/')        # 图 9.15 左侧
```

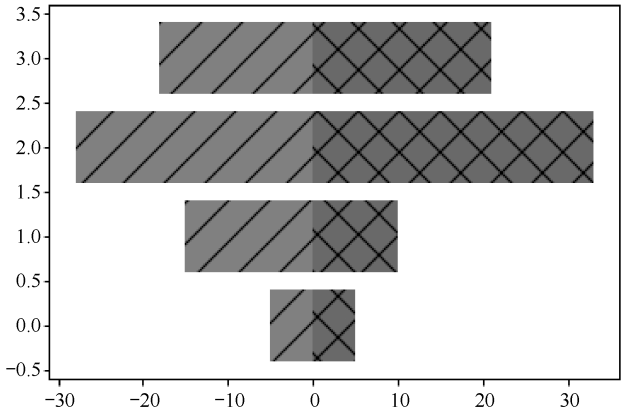


图 9.15　水平对比条形图

水平条形图显示时将 x 轴、y 轴交换，纵向是 x 轴，横向是 y 轴。将数据按从下往上的形式依次排列。本例中还使用了一个小技巧，即在绘制第二个水平条形图时提供的数据值是-data2，这样就在左侧的负半轴上显示出对比图形。

9.2.2　饼图

饼图用于显示数据集中各数据所占的百分比，它是最易理解的图形。最简单的饼图只需提供一组数值和标签，绘图库会自动计算各数值所占的比例。

```
rate = [1, 5, 3, 8]
labels = ['一月', '二月', '三月', '四月']
plt.pie(rate, labels=labels)              # labels 用于设置标注文字
plt.show()                                # 见图 9.16
```

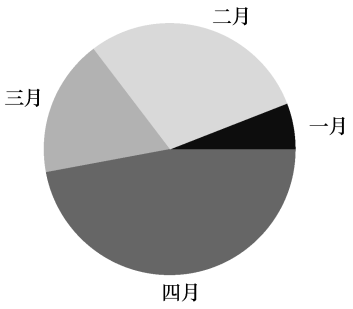


图 9.16　饼图

上面显示的饼图还不完善，如下代码可对其进一步加工以突出显示效果。

```
rate = [1, 5, 3, 8]
labels = ['一月', '二月', '三月', '四月']
plt.figure(figsize=(6,9))             # 设置图形大小
explode = (0.1, 0, 0, 0.05)           # explode 设置各部分分割出来的间隙
patches, ltext, ptext = plt.pie(rate, explode=explode, labels=labels,
                                autopct='%.1f%%', shadow=False, startangle=90)
# autopct:百分比数字的显示格式，%.1f%%表示保留一位小数
# shadow:是否有阴影
# startangle:起始角度。默认从 0 度逆时针开始为第一块，此处选择从 90 度开始（一月数据）
for  x  in  ltext:
    x.set_size(20)      # 设置标注文字大小
for  x  in  ptext:
    x.set_size(24)      # 设置百分比文字大小
plt.axis('equal')       # 设置 x/y 轴的单位长度相等
plt.legend()            # 显示图例，见图 9.17
```

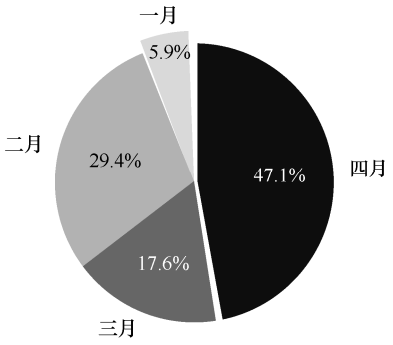


图 9.17　修饰后的饼图

上例中需要注意的几个地方如下：

（1）语句 explode = (0.1, 0, 0, 0.05)用于设置各部分从整圆中突显的比例。这里设置“一月”数据突显 0.1，“四月”数据突显 0.05。

（2）语句 patches, ltext, ptext=plt.pie()执行后返回一系列对象，其中 ltext 代表标注文字，ptext 代表百分比文字。因为默认的文字较小，所以对返回的这些对象设定文字大小，如程序中的 for 语句所示。

9.2.3 散点图

使用命令 plt.scatter()绘制的散点图在图上标记出每个点，根据散点图可以初步判断数据点的关系，如是否是线性相关，如图 9.18 所示。

```
np.random.seed(7)                          # 设置随机种子 7
x = np.random.uniform(-2, 3, 20)
y1 = 3 * x + 1 + 2 * np.random.random(20) # 生成直线点并加入随机值干扰
arg = np.polyfit(x, y1, 1)                 # 拟合直线，得到一次多项式的系数
f = np.poly1d(arg)                         # 利用系数构造拟合一次多项式
y2 = f(x)                                  # 根据拟合的多项式计算 y2 值
plt.scatter(x, y1)                         # 画原始数据的散点图，见图 9.18
plt.figure()                               # 产生新图
plt.scatter(x, y1)                         # 先画原始散点图，见图 9.19
plt.plot(x, y2, color='r')                 # 再画拟合直线
plt.text(1, 1, f, fontsize=16)             # 在坐标(1,1)处显示拟合的多项式 f
```

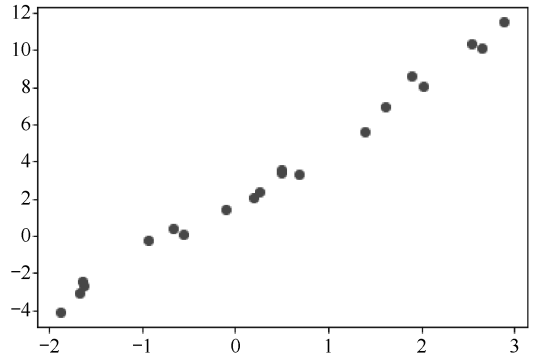


图 9.18 散点图

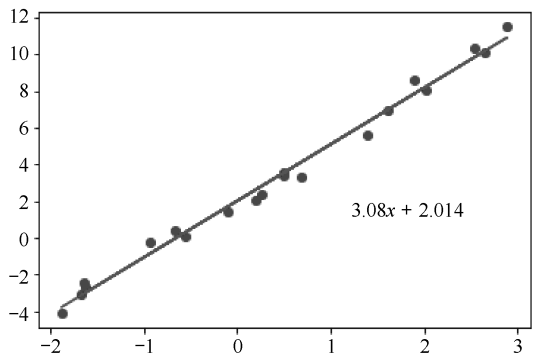


图 9.19 含拟合直线的散点图

上例中先画出散点图，观察发现这些点的分布类似直线，因此使用 NumPy 中的 poly1d()方法拟合直线（参阅 7.4.4 节），拟合公式为 $y = 3.08x + 2.014$。注意上例中的 plt.figure()执行后会创建一个新图形，后续的作图命令在新图形上作图。散点图适合数据量较小的情况，如果数据量很大，那么作图耗时会较长。

可以为散点图中的每个点单独设定颜色或大小，以便更好地区分数据。下面的代码将随机生成 20 个点，并赋予这些点不同的颜色和大小，输出如图 9.20 所示。设置时注意 colors 和 sizes 数组长度要和数据长度一致。代码中的 alpha=0.5 设置透明度，取值范围为[0,1]，1 表示不透明，颜色最深。各点的颜色值是一个随机小数，通过 cmap='viridis'参数自动映射为某种颜色。

```
N=20                                       # 总点数
rng = np.random.RandomState(7)             # 一个随机种子为 7 的随机数生成器
x = rng.randn(N)                           # 生成 N 个标准正态分布小数
y = rng.randn(N)
```

```
colors = rng.rand(N)
sizes = 1000 * rng.rand(N)            # 点的大小为随机值
plt.scatter(x, y, c=colors, s=sizes, alpha=0.5, cmap='viridis')
                                      # 设置点的颜色、大小、透明度
plt.colorbar()                        # 显示颜色条
```

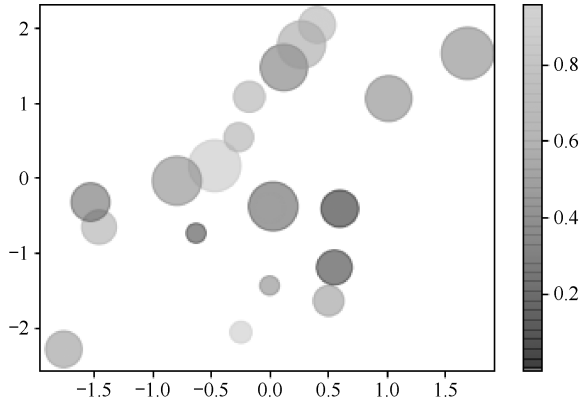


图 9.20　设置了散点大小和颜色的散点图

下面用鸢尾花 iris 数据集绘制两个散点图。这个数据集有 150 条记录，共三种鸢尾花。每条数据包含花萼（sepal）长度、宽度，花瓣（petal）长度、宽度，花的类型共 5 个数据项。机器学习中常用此数据集测试分类算法，计算机利用前 4 个数据项判断花的类型，然后与给定的花的类型对比，从而判断算法的优劣。

```
In: import  seaborn as sns                # 引入 sns
In: iris = sns.load_dataset('iris')       # 加载 iris 数据集，得到 DataFrame
In: iris.shape                            # 数据集大小
Out:  (150, 5)
In: iris.species.unique()                 # 有三类花，如下
Out: array(['setosa', 'versicolor', 'virginica'], dtype=object)
In: iris.head(2)                          # 观察数据，注意 species 已标注花类型
Out:   sepal_length  sepal_width  petal_length  petal_width  species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
```

要判断花的类型，只能从给定的数据出发。我们可将长度、宽度作为一组数据，分别标记出三种花的花萼和花瓣长/宽对比散点图，看看图形能否提示判断依据。散点可用颜色区分，但考虑到本书是黑白印刷的，因此用三种散点区分不同的形状。由于 plt.scatter()命令每次作图时只能使用相同的 marker 标记，所以下面的代码分三次画图，每次设定不同的 marker。

```
import  seaborn  as  sns
iris = sns.load_dataset('iris')           # 加载数据集
for  x  in  [0,1]:                        # 循环 2 次，分别作出花萼、花瓣的散点图
    if  x == 0:
        s = '花萼'
    else:
        s = '花瓣'
    plt.figure()                          # 创建新图
    plt.title('鸢尾花' + s + '长度-宽度散点图', fontsize=18)
    plt.xlabel(s + '长度', fontsize=16)
```

```
        plt.ylabel(s + '宽度', fontsize=16)
        # 下面用字典形式规定三种花的不同标记符号
        for  k, v  in  {'setosa':'^', 'versicolor':'v', 'virginica':'o'}.items():
            d = iris[iris.species==k]          # 每次筛选一种花，分三次完成
            plt.scatter(d.iloc[:, 2*x], d.iloc[:, 2*x+1], marker=v, s=120, label=k)
                                               # 指定不同的 marker
        plt.legend(fontsize=14)
```

绘图结果如图 9.21 和图 9.22 所示。

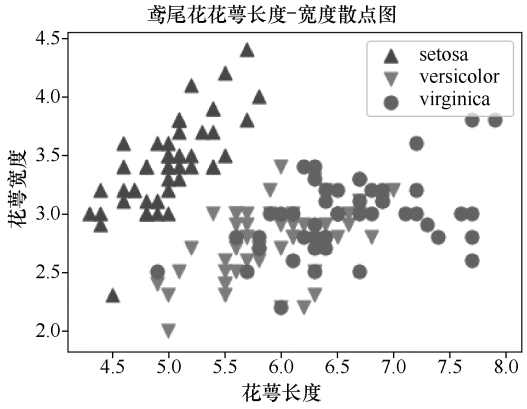


图 9.21　花萼长/宽散点图

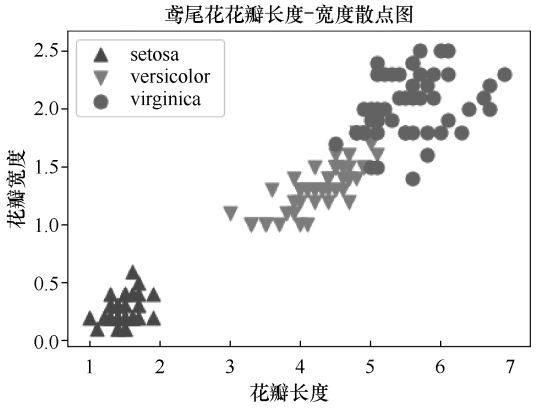


图 9.22　花瓣长/宽散点图

从图 9.21 和图 9.22 可以看出，三种花在图形上的区分度较大，尤其是花瓣散点图，这提示我们可以通过花瓣的长度、宽度判断花的类型。根据已知数据集计算出三类花的平均花瓣长度、宽度，以后遇到未知类型的数据时，与平均值比较即可以较高的正确率判断花的类型。

9.2.4　直方图

直方图（Histogram）又称质量分布图，它是统计学中常见的图形，主要用于表示数据的分布情况。图形的横轴表示数据区间段，纵轴表示每个区间段内数据的频数或频率。借助直方图可以看出数据分布的密集区间、稀疏区间，并猜测数据的可能分布。Matplotlib 中绘制直方图的函数是 plt.hist()。

```
arr = np.random.randn(1000)
plt.hist(arr,bins=10,density=0,edgecolor='b',facecolor='cyan', alpha=0.8)  # 见图 9.23
```

直方图的参数较多，只有第一个参数 arr 是必需的，后面的参数皆为可选参数。

- arr：用于绘制直方图的一维数组
- bins：直方图的柱数，默认为 10
- density：0 显示频数，1 显示频率（即频数/数据总个数）
- edgecolor：柱形的边线颜色
- facecolor：直方图颜色
- alpha：透明度

参数表中的 bins 指定划分的区间段数量，默认是等间隔的 10 个区间段。plt 按最小值到最大值等分为 10 个区间段，各个数据归类到对应的区间段，依此计算每个区间段的数据个数得到各区间段的频数。绘图时若 density=1，则纵坐标是频率；若 density=0，则纵坐标是频数。

```
arr = np.random.randn(1000)
```

```
# 自定义 bins 区间范围，见图 9.24
plt.hist(arr,bins=[-4,-2,0,1,3],density=1,facecolor='cyan',edgecolor='b',alpha=0.8)
plt.xticks(fontsize=14)              # 设置 x 轴标记文字大小
plt.yticks(fontsize=14)              # 设置 y 轴标记文字大小
```

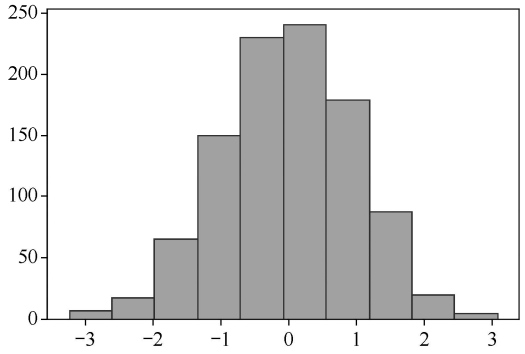


图 9.23　频数直方图

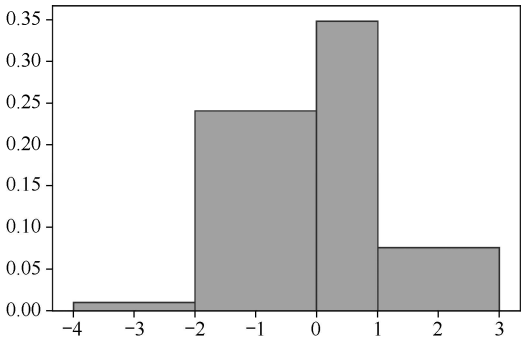


图 9.24　频率直方图（自定义 bins）

bins 参数也可自定义区间范围，上例中的 bins 是[-4,-2,0,1,3]，系统将统计各个区间的频率并作图，从图中可以看出[0,1]区间内含有约 35%的数据。

9.2.5　箱线图

箱线图（Box plot）又称盒式图或箱形图，也是统计学上的常见图形。该图用于反映数据的分布情况，一般包含若干关键的数值点，例如须线下限值、下四分位数（Q1）、中位数（median）、上四分位数（Q3）、须线上限值等。箱线图的详细解释请参阅统计学相关书籍，也可用命令“plt.boxplot?”查阅帮助文件。

```
arr = np.random.randn(150)           # 生成 150 个标准正态分布的随机小数
arr = np.append(arr, [-3.5, 4.1])    # 特意增加两个离群点
plt.boxplot(arr)                     # 画箱线图，见图 9.25
```

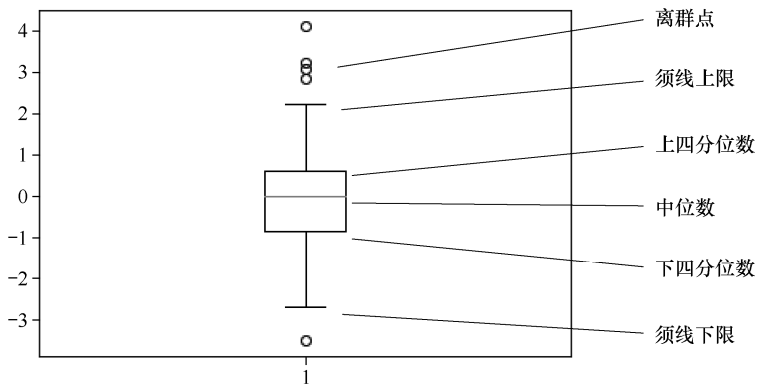


图 9.25　箱线图

下四分位数（Q1）、中位数、上四分位数（Q3）组成一个盒子。四分位距 IQR=Q3-Q1，即上四分位数与下四分位数之差构成盒子的长度，这个区间包含了 50%的数据点分布。

箱线图中定义最小观测值 min=Q1-1.5*IQR，最大观测值 max=Q3+1.5*IQR。大多数正常数据都应分布在区间[min, max]内，统计学上将位于区间[min, max]外的数据点称为离群点（异常点）。

因为离群点可能代表异常数据，所以在分析时要特别注意。

箱线图盒子中间的一条竖直线被称为胡须线（whisker）。考察剔除离群点后的正常数据，胡须线的下限取正常数据中的最小值，胡须线的上限取正常数据中的最大值，上限、下限各用一短横线表示，离群点单独用小圆圈标记。下面再来看一个箱线图的例子。

```
arr = [-3, 0, 5, 5, 6, 7, 8, 9]
print(np.percentile(arr, [25, 50, 75]))   # 输出的分位值为[3.75  5.5  7.25]
plt.boxplot(arr, showmeans=True)          # True 表示要显示平均值
plt.grid(axis='y')                        # 显示水平格线
plt.yticks(fontsize=14)                   # 见图 9.26
```

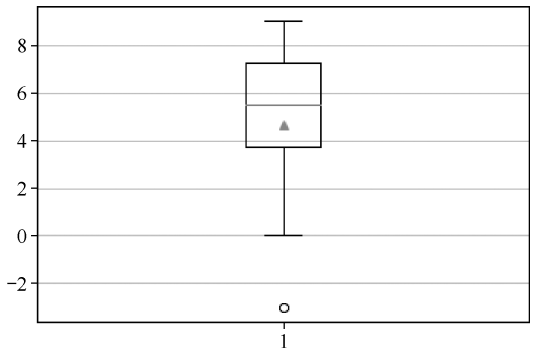


图 9.26　箱线图（显示均值）

上例中，通过 np.percentile(arr,[25,50,75])方法计算得到 arr 的 25%、50%、75%分位点，即 Q1、Q2、Q3，分别为 3.75、5.5、7.25，计算如下：

```
IQR = Q3 - Q1 = 7.25 - 3.75 = 3.5
min = Q1 - 1.5*IQR = 3.75 - 5.25 = -1.5
max = Q3 + 1.5*IQR = 7.25 + 5.25 = 12.5
```

正常数据的范围是[-1.5,12.5]，数据中的-3 是离群点，剔除-3 以后，正常的数据列表为[0,5,5,6,7,8,9]，所以胡须线的下限为 0，上限为 9。

作图时如果设置参数 plt.boxplot(arr,vert=False)，那么可以得到水平箱线图。

分析数据时，箱线图能帮助我们识别数据的特征，可以直观地显示离群点，容易判断数据集的数据离散程度和偏向（观察盒子的长度、上下隔间的形状及胡须的长度）。

9.2.6　其他图形

Matplotlib 还提供众多绘制其他图形的函数，例如：

- plt.axhline()：绘制水平线
- plt.axvline()：绘制垂直线
- plt.axhspan()：绘制水平区间带
- plt.axvspan()：绘制垂直区间带
- plt.errorbar()：绘制误差图

下面绘制水平线、垂直线、水平区间带、垂直区间带。

```
plt.xlim(0,1)                                          # 设 x 轴的坐标区域
plt.ylim(0,1)                                          # 设 y 轴的坐标区域
plt.axhline(y=0.2, xmin=0.1, xmax=0.5, color='g')      # 水平线，x 轴的范围[0.1,0.5]
```

```
plt.axhline(y=0.4, xmin=0.2, xmax=0.6, color='r')          # 水平线，x 轴的范围[0.2,0.6]
plt.axvline(x=0.8, ymin=0.1, ymax=0.4, color='r')          # 垂直线，y 轴的范围[0.1,0.4]
plt.axvline(x=0.9, ymin=0.2, ymax=0.5, color='b')          # 垂直线，y 轴的范围[0.2,0.5]
plt.axhspan(ymin=0.6, ymax=0.8, xmin=0.1, xmax=0.5)        # 水平区间带
plt.axvspan(xmin=0.7, xmax=0.8, ymin=0.6, ymax=0.9)        # 垂直区间带，见图 9.27
```

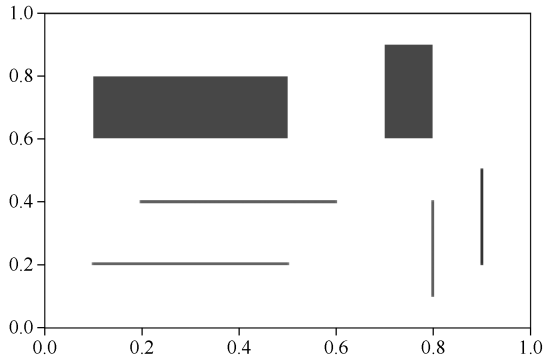


图 9.27　水平线、垂直线、水平区间带、垂直区间带

9.3　多图绘制

我们可以在一幅图上同时显示多个子图形，以便进行比较研究。前面介绍了基本的绘图函数，但如果想实现多子图绘制，就需要使用 Matplotlib 的 Figure 对象。下面绘制一个包含 4 幅子图的图形，如图 9.28 所示。

```
x = np.arange(1,100)
fig = plt.figure()                  # 创建一个 Figure 对象
ax1 = fig.add_subplot(221)          # 2 行 x2 列图形的第一个子图
ax1.plot(x, x)
ax2 = fig.add_subplot(222)          # 2 行 x2 列图形的第二个子图
ax2.plot(x, -x)
ax3 = fig.add_subplot(2, 2, 3)      # 2 行 x2 列图形的第三个子图
ax3.plot(x, x**2)
ax4 = fig.add_subplot(2, 2, 4)      # 2 行 x2 列图形的第四个子图
ax4.plot(x, np.log(x))
plt.show()                          # 见图 9.28
```

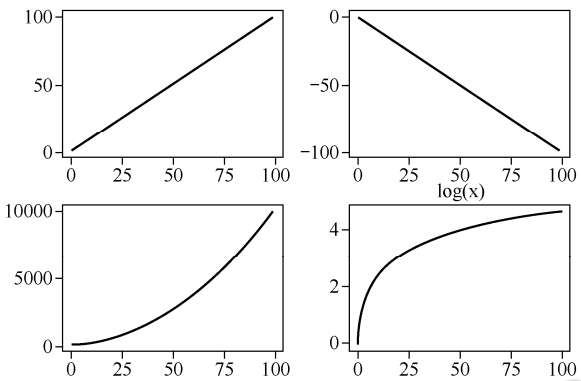


图 9.28　2×2 多子图示例

上例的代码中，fig=plt.figure()创建一个 Figure 对象，这个对象代表当前的整个图形区域，通过 plt.gcf()可以得到当前的 Figure 引用。绘图时若不显式地创建此对象，则 Matplotlib 会自动地创建 Figure 对象（前两节绘图时就自动创建了 Figure 对象）。创建 Figure 对象后，就可借助 fig.add_subplot()函数创建子图。

得到的 ax1、ax2 等变量是 AxesSubplot 类型的，这些变量可视为绘图的子区域。这些区域变量具有一系列绘图命令，语法风格和“plt.方法()”的风格类似。例如，plt.xlabel()对应于 ax.set_xlabel()、plt.xlim()对应于 ax.set_xlim()、plt.title()对应于 ax.set_title()等。在前述代码的 plt.show()的前面，增加如下几行代码就可给子图设置相应的标题，如图 9.29 所示。

```
plt.subplots_adjust(wspace=0.4, hspace=0.6)    # 调整各子图的水平、垂直间距
ax1.set_xlabel('x 轴', fontsize=14)            # 子图设坐标轴文字
ax2.set_ylabel('y 轴', fontsize=14)

ax3.set_title('x^2', fontsize=14)              # 子图设标题
ax4.set_title('log(x)', fontsize=14)
plt.show()                                     # 见图 9.29
```

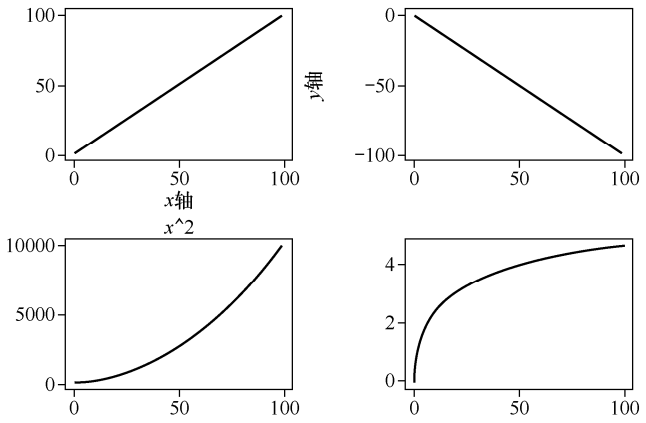


图 9.29　设置子图的标题和坐标轴

注意，上例中调用了 plt.subplots_adjust()方法来调整子图间的间距。如果不调整，那么坐标轴和标题文字会互相覆盖。

由于创建 Figure 和 subplot 对象都是很常见的绘图任务，因此绘图库提供了一个更方便的方法 plt.subplots()。

```
# 返回一个 Figure 和一个 2x3 的 axes 数组，sharex/sharey 表示各子图使用同一个 x,y 轴刻度
fig, axes = plt.subplots(nrows=2, ncols=3, sharex=True, sharey=True)
for  m  in  range(2):
    for  n  in  range(3):                      # 画 2x3 的随机直方图
        axes[m, n].hist(np.random.randn(500), bins=50, color='k', alpha=0.5)
plt.subplots_adjust(wspace=0, hspace=0)    # 子图间水平和垂直间距为 0，见图 9.30
```

代码行 fig, axes = plt.subplots(nrows=2, ncols=3)返回一个 Figure 和一个 axes 数组，注意 axes 数组的形状为(2,3)，其中包含 6 个子图，后续调用时使用形式 axes[m,n]。

创建子图时，还可用 plt.axes([left, bottom, width, height])方法直接指定子图的位置，指定时将窗口的横、纵长度视为 1，设定相应的位置、宽度、高度，如图 9.31 所示。

```
ax1 = plt.axes([0.05, 0.1, 0.4, 0.32])# 左下角 x 坐标、y 坐标、宽度、高度
ax2 = plt.axes([0.52, 0.1, 0.4, 0.32])
ax3 = plt.axes([0.05, 0.53, 0.87, 0.44])
ax1.grid(True)                                  # ax1 显示横纵网格线，图 9.31 左下方子图
ax2.xaxis.grid(color='r', linestyle='--', linewidth=1)
                                                # ax2 显示纵向网格线，图 9.31 右下方子图
ax3.yaxis.grid(color='b', linestyle='-', linewidth=2)
                                                # ax3 显示横向网格线，图 9.31 上方子图
```

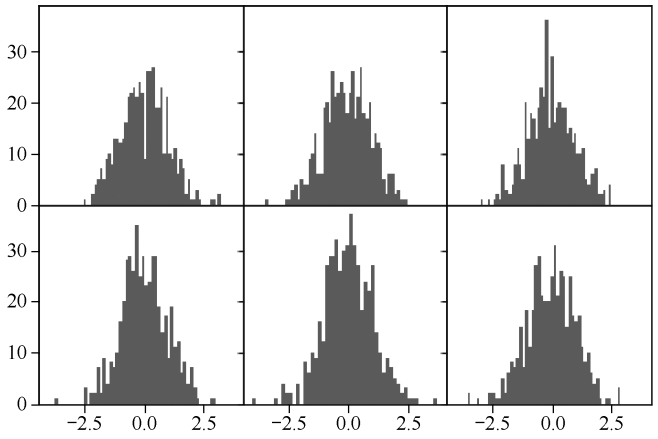


图 9.30　水平和垂直间距设为 0 的多个子图

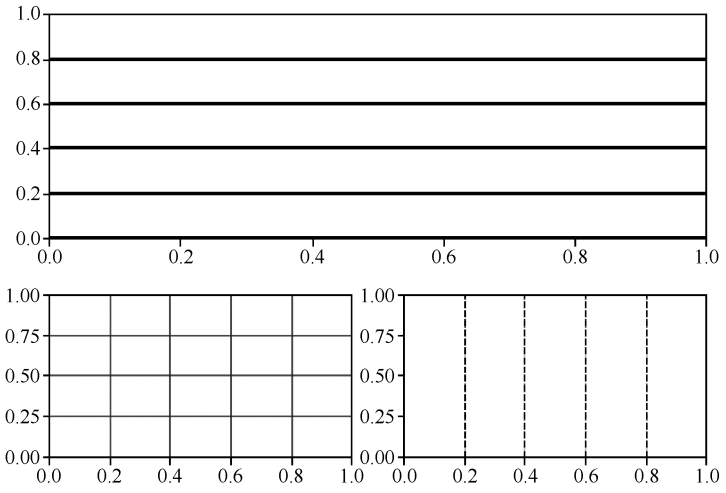


图 9.31　plt.axes()直接设定子图位置

下面使用对比子图展示常用的线型和标记符号。

```
def  axes_settings(fig, ax, title, ymax):       # 定义一个修饰 axes 的函数
    ax.set_xticks([])                           # 隐藏 x 轴刻度
    ax.set_yticks([])                           # 隐藏 y 轴刻度
    ax.set_ylim(0, ymax+1)                      # 设 y 轴的显示范围
    ax.set_title(title)                         # 设标题

x = np.linspace(-5, 5, 5)
```

```
y = np.ones_like(x)
fig, axes = plt.subplots(1, 2, figsize=(8,3))  # 产生 1x2 子图
linestyles = ['-', '-.', ':', '--']            # 线型
for  n, ls  in  enumerate(linestyles):
    axes[0].plot(x, y + n, color="blue", lw=2, linestyle=ls)
                                                # 在 axes[0]上绘制不同的线型

# 标记大小和颜色
sizecolors = [(4, "white"), (8, "red"), (12, "yellow"), (16, "lightgreen")]
for  n, (size, facecolor)  in  enumerate(sizecolors):
                                    # 在 axes[1]上绘制不同大小的标记和颜色
    axes[1].plot(x,y+n,'bo-',markersize=size,markeredgewidth=2,markerfacecolor=facecolor)

axes_settings(fig, axes[0], "线型", len(linestyles))
axes_settings(fig, axes[1], "标记大小/颜色", len(sizecolors))  # 见图 9.32
```

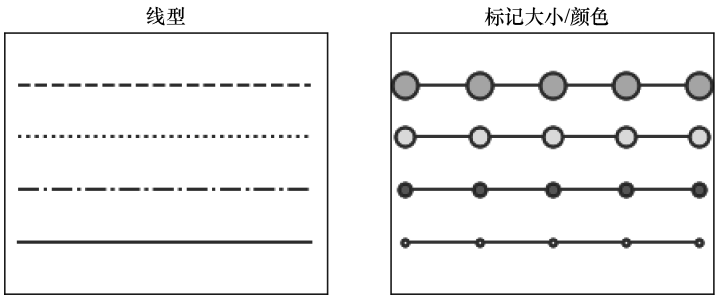


图 9.32　常用线型和标记符号演示

9.4　设置图形装饰项

Matplotlib 的图形有很多装饰项，可以设置标题、刻度、坐标轴标签文字等。设置方法主要有两种：plt.方法(参数表)和 axes.set_方法(参数表)。这些方法在调用时若不带参数，则返回当前的参数值，如 plt.xlim()返回当前 x 轴的绘图范围；在调用时若带参数，则设置参数值，如 plt.xlim(0,10)将 x 轴的显示范围设为 0 到 10。

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)                  # 创建一个 ax 对象
ax.plot(np.random.randn(1000).cumsum())        # 随机数累加并绘制折线图

ticks  = ax.set_xticks([0, 250, 500])          # x 轴上只显示 0,250,500 处的刻度
# 将上面三个刻度值用如下三个文字串显示
labels = ax.set_xticklabels(['数据一', '数据二', '数据三'], rotation=30, fontsize=14)

ax.set_title('随机数累加', fontsize=18)
ax.set_xlabel('X 轴', fontsize=18)
ax.set_ylabel('Y 轴', fontsize=18)             # 见图 9.33
```

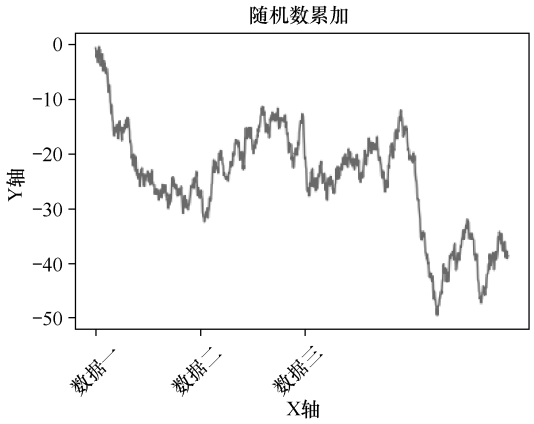


图 9.33　设置图形装饰项

9.4.1　添加注解

可以使用 plt.text()或 plt.annotate()方法在图形上添加注解文字。plt.text()的使用语法为 plt.text(x, y, '注解文字')，其中 x 和 y 是注解文字的坐标位置。

```
plt.axis([0, 5, 0, 20])                          # 设定显示范围 x 轴[0,5], y 轴[0,20]
plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'ro')      # 绘制 4 个红点

# 在(1.1, 12)位置用 LaTeX 语法显示 y=x^2
plt.text(1.1, 12, r'$y=x^2$', bbox={'facecolor':'yellow', 'alpha':0.2})
plt.text(1, 1.5, 'first')                        # 在(1, 1.5)处显示 first
plt.text(2, 4.5, 'second')                       # 在(2, 4.5)处显示 second

# 定义一个 box 修饰字典，设定各种格式（前景色、边线颜色、边框类型）
box = {'facecolor':'0.75', 'edgecolor':'k', 'boxstyle':'round'}
plt.text(3, 7.5, 'Third', bbox = box)            # 见图 9.34
```

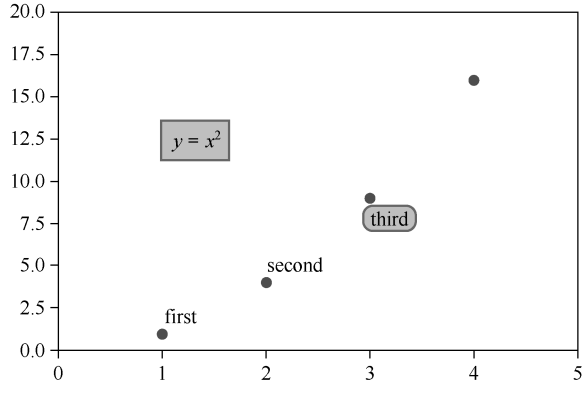


图 9.34　plt.text()标注示例

Matplotlib 支持 LaTeX 排版标记语言，可以方便地显示分数、根号、极限等各种数学表达式。LaTeX 标记由一对$符号包裹，内部以“\特殊字符串”的形式表达，详细情况可以访问网页 https://www.latex-project.org/或 https://matplotlib.org/users/mathtext.html。例如，$y=x^2$显示 x 的平方，$\frac{2}{3}$显示分数 2/3。下面是一些 LaTeX 表达示例，如图 9.35 所示。

```
plt.xlim([0, 10])
plt.ylim([0, 10])
plt.text(1, 9, r'希腊字母$\alpha > \beta$', fontsize=14)     # LaTeX 语法$标记符号$
plt.text(1, 7, r'下标$\alpha_i > \beta_i$', fontsize=14)
plt.text(1, 5, r'上标$s^2+y^3$', fontsize=14)
plt.text(1, 3, r'分数 $\frac{2}{3}$', fontsize=14)
plt.text(1, 1, r'平方根 $\sqrt{x^2 + y^2}$', fontsize=14)
plt.text(5, 9, r'求和符号$\sum_{i=0}^\infty x_i$', fontsize=12)
plt.text(5, 7, r'开 n 次方$\sqrt[3]{x}$', fontsize=16)
plt.text(5, 5, r'$sin(\frac{2\pi}{3})=\frac{\sqrt{3}}{2}$', fontsize=12)
plt.text(5, 3, r'积分$\int_a^b f(x)\mathrm{d}x$', fontsize=12)    # 见图 9.35
```

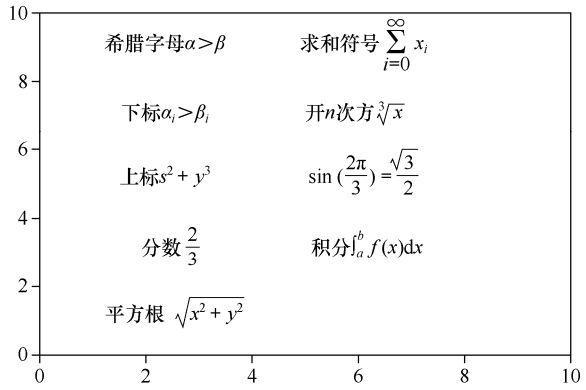


图 9.35 LaTeX 语法示例

除了 plt.text()方法，plt.annotate()也可在指定坐标处添加注解，并可设定指向箭头。

```
x = [20, 15, 32, 63, 70]
y = [75, 90, 130, 100, 80]
plt.axis([0, 80, 60, 140])                          # 设置 x/y 轴范围
labels = list('abcde')
plt.scatter(x, y)                                   # 绘制 5 个点
for  label, xpos, ypos  in  zip(labels, x, y):      # for 循环给每个点加标记
  plt.annotate(label, fontsize=16,
            xy=(xpos, ypos),              # 指明标记文字的坐标位置，见图 9.36
            xytext=(6, -5),               # 但要有轻微偏离：向右(6)，向下(-5)
            textcoords='offset points')   # 此属性表示 xytext 是相对 xy 偏离的
```

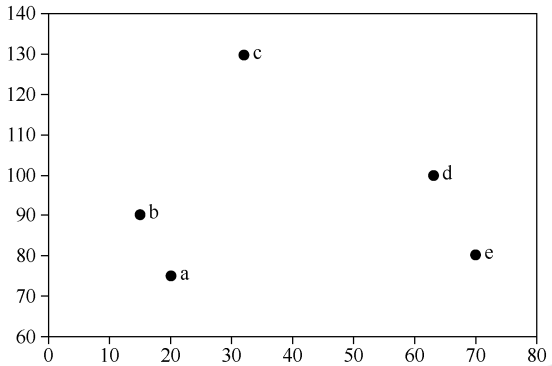


图 9.36 plt.annotate()添加注解文字

再看一个 plt.annotate()的例子，本例加上了箭头指示。

```
x = [20, 15, 32, 63, 70]
y = [75, 90, 130, 100, 80]
x = np.linspace(-np.pi*2, np.pi*2,100)
y = np.sin(x)
plt.plot(x, y)
plt.annotate('顶部',
     xytext=(3.5, 1),       # 文字位置
     xy=(np.pi/2, 1),       # 箭头位置的坐标
     arrowprops={'facecolor' : 'black', 'shrink' : 0.05}) # 箭头外观
plt.annotate('底部',
     xytext=(-np.pi/2, -0.5),    # 文字位置
     xy=(-np.pi/2, -1),          # 箭头位置的坐标，见图 9.37
     arrowprops={'facecolor' : 'blue', 'shrink' : 0.2})  # 箭头外观
```

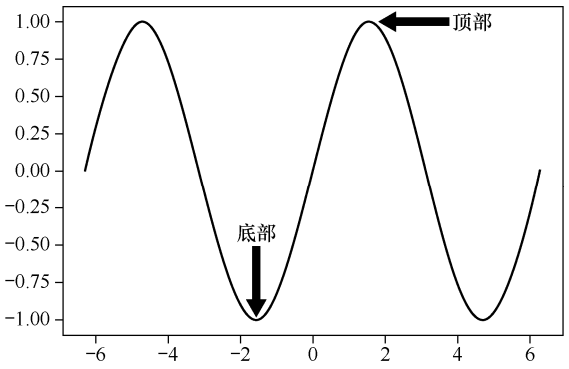


图 9.37　plt.annotate()箭头标注示例

绘图命令含有很多的参数，部分参数用文字很难解释，读者动手试一试就会理解。

9.4.2　设置坐标轴

对坐标轴的修饰主要包括设置原点位置、设置坐标轴上的刻度间距、设置标记文字等。下面来看一个设置原点位置的例子。

```
def  format_spines():                              # 将坐标原点设置在图中央
    ax = plt.gca()                                 # gca 代表当前坐标轴
    ax.spines['right'].set_color('none')           # 隐藏坐标轴右侧的边线
    ax.spines['top'].set_color('none')             # 隐藏坐标轴顶部的边线
    ax.spines['bottom'].set_position(('data', 0))  # 下边线移到 0 位置，X 轴
    ax.spines['left'].set_position(('data', 0))    # 左边线移到 0 位置，Y 轴

x = np.linspace(-np.pi, np.pi, 100)
plt.plot(x, np.sin(x), color='r', label='sin(x)')
plt.plot(x, np.cos(x), color='b', label='cos(x)')
xlabels = (r'$-\pi$', r'$-\pi/2$', r'$+\pi/2$', r'$+\pi$')   # LaTeX 语法
plt.xticks((-np.pi, -np.pi/2, np.pi/2, np.pi), xlabels)      # 修改 x 轴刻度显示
plt.yticks([-1, -0.5, 0, 0.5, 1])
plt.legend(loc='upper left')                       # 见图 9.38
```

```
plt.tick_params(axis='both', labelsize=14)     # 设定轴上刻度的文字大小
format_spines()                                # 调用自定义函数设坐标轴
```

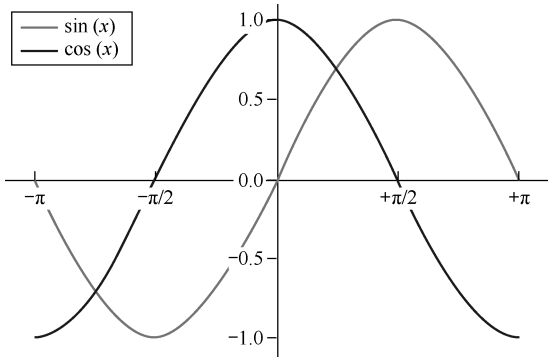


图 9.38　设置坐标原点位置的图形

图形默认的坐标原点在左下角。如果需要将坐标原点放在图形中心，那么可以先将轴的右侧边线和顶部边线的颜色设为 none（隐藏），再将左侧的边线（即 y 轴）移到 0 刻度位置，将下侧的边线（即 x 轴）移到 0 刻度位置。

坐标轴上的刻度默认是数字，但可使用命令 plt.xticks([坐标点], [标记文字])将对应的坐标点标识修改为需要显示的字符。例如，plt.xticks([2,5,6], ['A','B','C'])就将 x 轴上的原来的数字标记全部清空，只在原来的位置 2,5,6 显示 A,B,C。如果方法调用时不提供标记文字，那么坐标轴上就只显示 2,5,6。

语句 plt.tick_params(axis='both', labelsize=14)可设置坐标轴上数字刻度的文字大小。

9.4.3　填充颜色和显示图片

可以使用 plt.fill_between()方法填充颜色以增强图形的视觉效果，如图 9.39 所示。

```
def  fill_sin():                               # 自定义的绘图函数
    n = 256
    X = np.linspace(-np.pi, np.pi, n)
    Y = np.sin(2 * X)
    plt.plot(X, Y + 1, color='b')              # 上方的蓝色 sin 线
    # 表示在 X 范围内，[1,Y+1]这段 Y 轴填充颜色
    plt.fill_between(X, 1, Y + 1, color='b', alpha=0.3)

    plt.plot(X, Y - 1, color='k')              # 下方的黑色 sin 线
    # 条件填充，(Y - 1) > -1 时填蓝色
    plt.fill_between(X, -1, Y - 1, where=(Y - 1) > -1, color='b', alpha=0.3)
    # 条件填充，(Y - 1) < -1 时填红色
    plt.fill_between(X, -1, Y - 1, where=(Y - 1) < -1, color='r', alpha=0.3)
    plt.xlim(-np.pi, np.pi)                    # 设定 x 轴的范围
    plt.ylim(-2.5, 2.5)                        # 设定 y 轴的范围

plt.figure(figsize=(6, 4))                     # 设图形大小，见图 9.39
fill_sin()                                     # 调用自定义绘图函数
plt.tight_layout()                             # 自动调整图形参数，使之填充整个图像区域
```

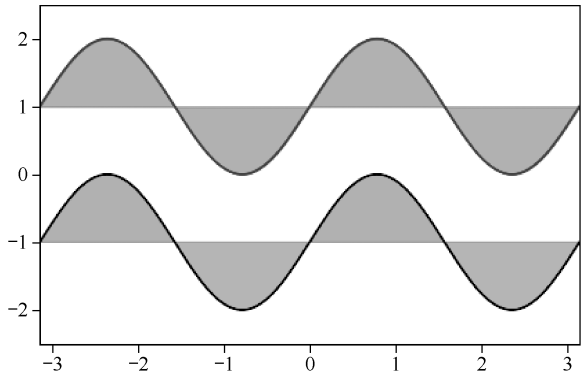


图 9.39　颜色填充图

plt.fill_between()方法用于填充颜色，其参数形式为 fill_between(x 轴区域，y 轴起点,y 轴终点，color='填充色'，alpha=透明度)。该方法在指定的 x 轴范围内，从 y 轴起点填充到 y 轴终点，并可设定填充的条件，只有满足条件时才填充。

plt 不仅可以绘制图形，而且可以显示图片文件。要显示图片文件，就要先安装 PIL 图形库。在命令行上执行 pip install pillow 或 conda install pillow，系统会自动下载并安装 PIL 库。安装成功后，执行下面的命令可显示图片。

```
from PIL import Image                # 导入 Image
import matplotlib.pyplot as plt
img = Image.open('tu.jpg')           # 将图片 tu.jpg 读入数组，图片文件应在当前目录中
print(img.size)                      # 查看图片数组大小
plt.axis('off')                      # 不显示坐标轴
plt.imshow(img)                      # 显示图片
```

显示图片的另一种方法是直接引入 IPython.display 中的 Image 函数，示例如下。

```
In: from IPython.display import Image as img
In: img(r'e:\pic\tu.jpg')            # 显示图片 e:\pic\tu.jpg
```

9.5　使用 Pandas 绘图

前几节讨论了 Matplotlib 中的绘图函数，其实 Pandas 中已融入了对 Matplotlib 的支持，可以直接利用 Pandas 提供的方法作图。Series 和 DataFrame 对象都带有一个 plot 方法，调用时指定图形类型即可。下面演示用 Pandas 作图的方法。

```
import pandas as pd
import numpy as np
import numpy.random as rnd
import matplotlib.pyplot as plt
rnd.seed(7)
x = np.arange(5)
df = pd.DataFrame(rnd.randint(1, 15, 12).reshape(4, 3), columns=list('ABC'))
df.index = ['一', '二', '三', '四']
df.plot(kind='bar', rot=0)      # rot=0 表示文字"一二三四"的旋转角度为 0，见图 9.40
```

程序会产生 12 个随机整数，创建一个 4×3 的数据框，数据如下。

```
In:df
```

```
Out:
    A   B   C
一  5  10   7
二  4   4   8
三  8  13  10
四  8   9  10
```

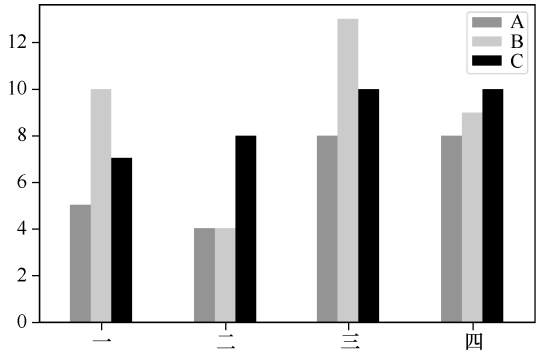


图 9.40　Pandas 绘制簇状柱形图

语句 df.plot(kind='bar', rot=0)指定绘制一个柱形图。Pandas 自动以每行为一组数据，每组数据中含 3 列数据，绘制 4 组柱形对比图。列索引名"ABC"成为图例，行索引"一二三四"成为 x 轴上的标记。rot=0 指定标记文字旋转 0 度，不设定旋转角度值时，文字默认旋转 90 度。

参数 kind='类型'用于指定作图的类型，常用的类型有 line（折线图）、bar（柱形图）、barh（水平条形图）、pie（饼图）、kde（密度图）、hist（直方图）等。不指定类型时，默认为折线图。

```
df.B.plot(ylim=[4, 15])                                    # 默认折线图
plt.xticks(np.arange(len(df.index)), df.index)             # 设置 x 轴刻度标识文字
for  x, y  in  zip(np.arange(len(df.B)), df.B):            # 标注每个点的数值
    plt.text(x, y+0.4, y, ha='center', va= 'bottom', fontsize=12)
```

上面的代码在绘制折线后，用 plt.text()为每个数据点加上数据值标注。标注时指定坐标位置为(x, y+0.4)，即将数值标注在比此点的 Y 值高 0.4 的位置。

下面再分别作一个饼图和一个堆积柱形图。

```
df.A.plot(kind='pie', title='A 列数据', fontsize=16)     # 以 A 列数据作饼图，见图 9.41
df.plot(kind='bar', stacked=True, rot=20)      # stacked=True 堆积柱形图，见图 9.42
```

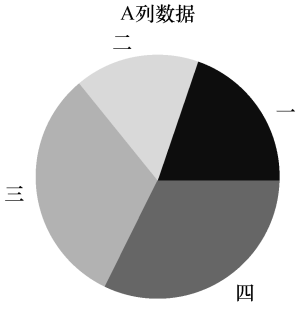


图 9.41　Pandas 绘制饼图

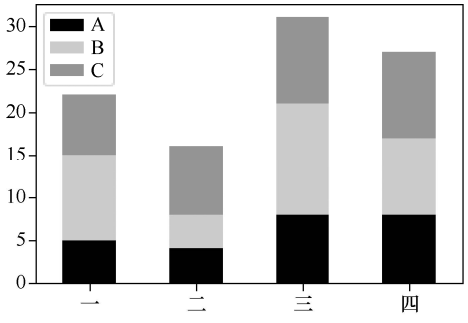


图 9.42　Pandas 绘制堆积柱形图

df.plot(kind='bar')也可写为 df.plot.bar()，它们的效果相同。

df.plot()方法中有很多参数，类似前面介绍过的 plt.plot()中的参数，如表 9.7 所示。

表 9.7　df.plot()的常用参数表

参 数 名	描　述
label	标记文字
style	格式符号，如'bo—'
alpha	透明度，取值(0,1)
kind	图形类型（'area', 'bar', 'barh', 'density', 'hist', 'kde', 'line', 'pie'）
rot	刻度标记旋转角度 0～360
xticks	x 轴上的刻度
yticks	y 轴上的刻度
xlim	x 轴的显示范围，如[-2, 3]
ylim	y 轴的显示范围
grid	显示网格（True/False）
use_index	是否使用索引作为标记
title	图形的标题

绘图常用参数示例如图 9.43 所示。

```
df.plot(style='bo-', xticks=[0, 1, 2, 3, 4], yticks=np.arange(0, 15,3), ylim=[3,15])
```

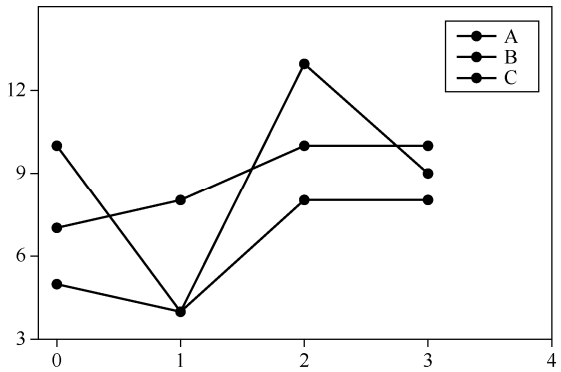


图 9.43　Pandas 绘图常用参数示例

9.6　本章小结

本章介绍了 Matplotlib 绘图模块，语法格式分为“plt.命令(参数表)”和“ax.set_命令(参数表)”两种。图形显示模式有独立窗口模式和嵌入模式。常见的绘图命令及所绘图形类型如下：plt.plot()，折线图；plt.bar()，柱形图；plt.barh()，水平条形图；plt.pie()，饼图；plt.scatter()，散点图；plt.hist()，直方图；plt.boxplot()，箱线图。图形中要显示中文时，需要修改参数或设定中文字体。

plt.plot(x,y,color='r',marker='o',ls=':')或 plt.plot(x,y,'ro:')设置颜色、标记符号、线型等选项。

plt.title()设置标题，plt.xlim()设置 x 轴的范围，plt.xlabel()设置 x 轴的标识，plt.xticks()设置 x 轴的刻度标记，plt.legend()显示图例，fig.add_subplot()创建多个子图，plt.text()在指

定坐标处标注文字，plt.annotate()标注箭头，plt.imshow()显示图片，plt.savefig()保存图片。

Pandas 可以绘制各类图形，调用形式为 df.plot(kind='图类型')。

习题

1. 在同一幅图上绘制区间[-2π,2π]内的正弦曲线和余弦曲线。
2. 自行下载一个股票数据文件，用 Pandas 读入数据，绘制某段时间内的收盘价折线图。
3. 一幅图上分 4 个子图，分别绘制圆形、正方形、三角形和直线。
4. 随机产生一组具有线性关系的点的坐标，绘制散点图。
5. 用数据集[2,5,8,6]绘制饼图，要加上百分比，并突显数据项 8。
6. 绘制曲线 $y = x^2 + 2x + 1$，要求坐标原点在图形中央。
7. plt 可以绘制哪些类型的图形？
8. 箱线图的须线上限和下限如何确定？
9. 如何绘制柱形图、水平条形图、堆积柱形图？
10. Matplotlib 的图形输出有哪两种方式？
11. 写出 plt 命令，将 x 轴的显示范围设为[1,5]，将 y 轴的显示范围设为[0,10]，将标题设为“Title”，并且只显示水平网格线。
12. 中文和负号显示不正常时，应设置什么参数？

第 10 章　Tushare 财经数据接口

本章首先介绍从 Tushare 平台获取各类财经数据的基本方法，然后介绍 Tushare 包的内建接口函数及其使用方法、用 Python 程序实现股票收盘价格与成交量时间序列数据的可视化（绘制恰当的图形以展示序列数据变化的特征）方法，最后介绍一种以公司定期财务报告数据的关键指标为依据来选择优质公司股票的 Python 实现方法。

10.1　财经数据接口 Tushare 简介

目前 Web 上存在众多提供常用财经数据的平台资源，其中 Tushare 网站是一个免费且适合 Python 开发者使用的财经数据平台。Tushare 平台可提供的财经数据涵盖我国宏观经济、国内股票市场各类指数、国内上市公司股票交易数据、上市公司定期财务报告及国内财经新闻等诸多类别的数据。

使用 Tushare 平台数据的 Python 开发者需要在本机安装 Tushare 包（如果本机处于连网状态，那么可在 Anaconda 集成环境的命令行操作界面执行 `pip install tushare` 命令完成安装）才能利用 Tushare 包的内建函数获取平台提供的财经数据。Tushare 内建函数返回的数据格式都是 Pandas 的 `DataFrame` 类型，所以比较方便利用 Pandas、NumPy、Matplotlib 等程序包提供的操作工具对这些函数的返回数据进行处理。

本章的所有程序均在 Spyder 3.3.6 环境下调试通过，其中 Tushare 内建函数及其输入参数有多种形式可供选择。建议读者参阅 Tushare 官网 http://tushare.org/index.html 提供的函数说明，深入了解这些常用财经数据接口函数的参数含义及其使用方法。

10.1.1　宏观经济数据

如果在本机安装了 Tushare 包，那么就可利用 Tushare 的内建函数从 Tushare 平台获取多种国内宏观经济数据，如多个时期的货币供应量、存款准备金率、存贷款利率、国内生产总值、居民消费价格指数及工业品出厂价格指数等。若在 IPython 操作界面执行如下操作，则可获取我国最近 30 年来各时期的货币供应量数据：

```
In [1]: import  tushare  as  ts
In [2]: df = ts.get_money_supply()
In [3]: df.head()
Out[3]:
    month    m2       m2_yoy       m1    ... sd sd_yoy       rests    rests_yoy
0  2019.6  1921360.19  8.50    567696.18 ... 790201.11  --  201300.14   --
1  2019.5  1891153.70  8.50    544355.64 ... 778725.61  --  203181.67   --
2  2019.4  1884670.33  8.50    540614.60 ... 776276.48  --  202474.45   --
3  2019.3  1889412.14  8.60    547575.54 ... 782606.12  --  200214.99   --
4  2019.2  1867427.45  8.00    527190.48 ... 773741.13  --  209785.79   --
[5 rows x 17 columns]
```

函数 `get_money_supply()`是 Tushare 包的内建函数。该函数返回的数据是 `DataFrame` 类的结

构数据，所以保存函数返回值的变量 df 具有多行多列数据，其中每行表示不同时期对应各项指标的数据。执行 df.head()命令可以显示 df 保存的前 5 行数据，执行 df.head(0)命令可以显示返回数据的所有列名称，每列数据的含义说明如表 10.1 所示。

```
In [4]:df.head(0)
Out[4]:
Empty DataFrame
Columns: [month, m2, m2_yoy, m1, m1_yoy, m0, m0_yoy, cd, cd_yoy, qm, qm_yoy, ftd,
          ftd_yoy, sd, sd_yoy, rests, rests_yoy]
Index: []
```

表 10.1 货币供应量接口参数

列 名 称	代表的内容	列 名 称	代表的内容
month	统计时间	qm	准货币（亿元人民币）
m2	货币和准货币（广义货币 M2）（亿元人民币）	qm_yoy	准货币同比增长（%）
m2_yoy	货币和准货币（广义货币 M2）同比增长（%）	ftd	定期存款（亿元人民币）
m1	货币（狭义货币 M1）（亿元人民币）	ftd_yoy	定期存款同比增长（%）
m1_yoy	货币（狭义货币 M1）同比增长（%）	sd	储蓄存款（亿元人民币）
m0	流通中现金（M0）（亿元人民币）	sd_yoy	储蓄存款同比增长（%）
m0_yoy	流通中现金（M0）同比增长（%）	rests	其他存款（亿元人民币）
cd	活期存款（亿元人民币）	rests_yoy	其他存款同比增长（%）
cd_yoy	活期存款同比增长（%）		

利用 get_money_supply()接口函数可以获得自 1978 年至 1995 年每年的年度货币供应量数据，以及自 1996 年 1 月至当前的每年每个月的货币供应量数据。

Tushare 平台支持的主要宏观经济数据接口函数（见表 10.2）都不需要提供任何输入参数就可以获取相应的数据，返回的数据包含自有公开数据以来的全部数据。由于不同接口函数返回的数据参数差别较大，因此请参考 Tushare 网页 http://tushare.org/macro.html 中的相关内容了解这些函数的细节。

表 10.2 宏观经济数据的接口函数

接口名称	功 能	接口名称	功 能
get_deposit_rate()	存款利率	get_gdp_year()	国内生产总值（年度）
get_loan_rate()	贷款利率	get_gdp_quarter()	国内生产总值（季度）
get_rrr()	存款准备金率	get_cpi()	居民消费价格指数
get_money_supply()	货币供应量	get_ppi()	工业品出厂价格指数
get_money_supply_bal()	货币供应量（年底余额）		

利率（也称利息率）是单位货币（元）在指定时间内的利息水平，其值是指定时间内所产生的利息额与本金的比率。根据计量的期限标准不同，利率的表现形式有年利率、月利率和日利率等。Tushare 针对不同种类的利率提供了相应的接口函数以供开发者获取所需的利率数据。利用存款利率函数 get_deposit_rate()和贷款利率函数 get_loan_rate()可以分别获取我国自 1989 年以来中国人民银行发布的各期存款利率和贷款利率数据。在 IPython 操作界面执行如下命令可以获取各时期的存款利率：

```
In [1]: import  tushare  as  ts
In [2]: df = ts. get_deposit_rate()
```

```
In [3]: df.head()
Out[3]:
date        deposit_type           rate
0  2015-10-24       定活两便(定期)        --
1  2015-10-24   定期存款整存整取(半年)   1.30
2  2015-10-24   定期存款整存整取(二年)   2.10
3  2015-10-24  定期存款整存整取(三个月)  1.10
4  2015-10-24   定期存款整存整取(三年)   2.75
```

以类似的方法使用函数 get_loan_rate()可以获取我国自 1990 年以来各时期的贷款利率数据。此外，还可以从 Tushare 平台获取一些专业性更强的利率数据，如 Shibor 利率数据、Shibor 报价数据、LPR 贷款基础利率数据、Libor 利率数据和 Hibor 利率数据等。

上海银行间同业拆放利率（Shanghai Interbank Offered Rate，Shibor）是以位于上海市的全国银行间同业拆借中心为技术平台计算、发布并命名，由信用等级较高的银行组成报价团自主报出的人民币同业拆出利率计算确定的算术平均利率。每个交易日要根据各报价行的报价（剔除最高、最低各 4 家报价）进行算术平均计算后得出每一期限品种的 Shibor 利率，并于 11:00 对外发布。目前对社会公布的 Shibor 品种包括隔夜、1 周、2 周、1 个月、3 个月、6 个月、9 个月及 1 年期的利率。获取指定时期内 Shibor 利率的操作方法如下：

```
In [1]: import  tushare  as  ts
In [2]: pro = ts.pro_api()
In [3]: df = pro.shibor(start_date='20180601', end_date='20190729')
In [4]:df.head()
Out[4]:
     date     on     1w     2w     1m     3m     6m     9m     1y
0  20190729  2.629  2.657  2.690  2.600  2.631  2.706  3.016  3.098
1  20190726  2.636  2.647  2.688  2.597  2.629  2.704  3.016  3.096
2  20190725  2.628  2.642  2.663  2.594  2.625  2.704  3.015  3.096
3  20190724  2.481  2.561  2.631  2.590  2.625  2.703  3.016  3.096
4  20190723  2.562  2.634  2.765  2.591  2.628  2.703  3.017  3.097
```

这个利率接口函数是 Tushare 平台提供的 pro 接口函数，需要利用 pro_api()命令才能获得 Shibor 利率数据对应的 pro 接口，然后利用该 pro 接口调用 shibor()方法获得指定日期内的利率数据。pro 接口的使用者需要在平台 https://tushare.pro/register 注册，并在运行环境下设置用户凭证信息后才能正常使用 pro 接口。具体操作过程包含如下 7 个步骤：

① 登录网页 https://tushare.pro/register，注册 Tushare 社区用户。

② 注册用户登录 https://tushare.pro/login 进入 Tushare 社区，然后依次执行如下三步操作来提取用户凭证信息：

第 1 步，用户在网站登录成功后，将鼠标移到页面右上角的用户名称位置，单击其下拉菜单中的“个人主页”选项进入“用户中心”。

第 2 步，用户在“用户中心”页面中单击如下的“接口 TOKEN”选项卡。

第 3 步，单击右侧“睁开眼睛图标”显示如下的 token 凭证信息，复制文本框中的全部内容。

③ 使用命令 pip install tushare 在本机安装 Tushare 包。

④ 在 IPython 操作界面执行导入 Tushare 包命令 import tushare as ts。

⑤ 利用 Tushare 包的内建函数 set_token()以如下方式设置本机用户的 token 凭证信息。

```
In [3]: ts.set_token('a56336[redacted]2068')
```

用户凭证信息在本机用户第一次调用 pro 接口前设置好后，通常是长期有效的。此后，若系统提示用户凭证信息缺失，则要按照这个方法重新设置凭证信息。

⑥ 使用命令 pro = ts.pro_api()初始化 pro 接口。

如果 set_token('用户 tushare token')无效或不想将 token 保存本地，那么可以在初始化接口时直接设置 token: pro_api('用户 token')。

⑦ 数据调取。完成前 6 项操作后，用户才能调用 pro 接口函数获取相应的数据。

其他的利率 pro 接口函数如表 10.3 所示。

表 10.3 其他的利率 pro 接口函数

接口名称	描 述	接口名称	描 述
pro.shibor()	上海银行间同业拆放利率	pro.Libor()	伦敦同业拆借利率
pro.shibor_quote()	Shibor 报价数据	pro.Hibor()	香港银行同业拆借利率
pro.shibor_lpr()	LPR 贷款基础利率		

使用表 10.3 所列的利率接口函数时，也可以设置函数的输入参数，但一般情况下会省略函数的输入参数来获取函数默认的返回值。需要注意的是，这些接口函数的输入参数项设置并不完全相同，其中 shibor()、shibor_lpr()和 Hibor()三个接口的输入参数都是 date、start_date 和 end_date，但 shibor_quote()接口的输入参数包括 date、start_date、end_date 和 bank，其中的参数 bank 应以中文给出银行名称，如'建设银行'。Libor()接口函数的输入参数包括 date、start_date、end_date 和 curr_type，其中 curr_type 用于指定货币代码（USD，美元；EUR，欧元；JPY，日元；GBP，英镑；CHF，瑞士法郎。默认是 USD）。

10.1.2 股票行情数据

用户从 Tushare 平台可以获取上海证券交易所与深圳证券交易所全部上市公司的股票交易数据，也可以获取这两个市场的各类股票指数（如上证综合指数、深证成份指数、创业板指数、沪深 300 指数和中小板指数等）数据。无论是单只股票的交易数据还是沪深两市的指数数据都属于时间序列数据，这些数据的时间周期都有日、周、月等周期，每个周期的数据包括周期内的最高价格、最低价格、开市价格、收市价格、成交量及涨跌幅度等股票交易数据。

Tushare 平台提供的股票数据包括历史行情数据和实时交易数据。通常用于获取股票历史行情数据的函数是 get_hist_data()。例如，依次执行下列命令可以获取股票代码为 600848 的历史日线行情数据：

```
In [1]: import tushare as ts
In [2]: df = ts.get_hist_data('600848', ktype = 'D')
In [3]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 522 entries, 2019-02-13 to 2016-08-15
Data columns (total 13 columns):
open            522 non-null float64
high            522 non-null float64
close           522 non-null float64
low             522 non-null float64
volume          522 non-null float64
price_change    522 non-null float64
p_change        522 non-null float64
ma5             522 non-null float64
ma10            522 non-null float64
ma20            522 non-null float64
v_ma5           522 non-null float64
v_ma10          522 non-null float64
v_ma20          522 non-null float64
dtypes: float64(13)
memory usage: 57.1+ KB
```

这表明函数返回了 522 条关于股票 600848 的日线行情数据，其中参数 ktype='D'表示获取的行情周期为“日”。函数返回数据 df 的列名称及数据组织形式可用 head()方法查看。

```
In [4]:df.head(0)
Out[4]:
Empty DataFrame
Columns: [open, high, close, low, volume, price_change, p_change, ma5,
        ma10, ma20, v_ma5, v_ma10, v_ma20]
Index: []
```

这些列数据的名称依次表示对应交易日一天中该股票的开市价格、最高的交易价格、收市价格、最低的交易价格、全天累计成交的股票数量（手，1 手等于 100 股）、交易价格涨跌数量（元）、交易价格涨跌幅度（%）、收市价格的 5 天移动平均值、10 天移动平均值、20 天移动平均值，最后三项依次是日成交量的 5 天移动平均值、10 天移动平均值及 20 天移动平均值。特别地，返回值是以交易日期 date 为索引构建的 Dataframe 结构，而且以时间倒序的方式排列全部交易数据。用 tail()方法查看最后 5 行数据如下：

```
In [5]: df.tail()
Out[5]:
            open   high  close   low   ...  ma20    v_ma5     v_ma10      v_ma20
date                                    ...
2017-02-13 20.38  20.98  20.79  20.27  ... 20.690   63978.98  63978.98   63978.98
2017-02-10 20.72  20.81  20.45  20.42  ... 20.665   65581.14  65581.14   65581.14
2017-02-09 20.52  20.82  20.67  20.42  ... 20.737   74643.62  74643.62   74643.62
2017-02-08 20.55  21.10  20.65  20.30  ... 20.770   91145.38  91145.38   91145.38
2017-02-07 19.97  21.20  20.89  19.68  ... 20.890  114606.50  114606.50  114606.50
```

这些数据表明 df 保存了这只股票自 2017 年 2 月 7 日至当前各交易日发生的股票交易数据，时间周期为“日”，也就是人们所说的日线数据。函数 get_hist_data()的输入参数名称及其表示的含义如表 10.4 所示。

表 10.4 函数 get_hist_data()的参数名称及其含义

参数名称	含 义 描 述
code	股票代码，即 6 位数字代码，或者指数代码（sh=上证指数，sz=深圳成指，hs300=沪深 300 指数，sz50=上证 50，zxb=中小板，cyb=创业板）
start	开始日期，格式 YYYY-MM-DD
end	结束日期，格式 YYYY-MM-DD
ktype	数据类型，D=日 k 线，W=周，M=月，5=5 分钟，15=15 分钟，30=30 分钟，60=60 分钟，默认为 D
retry_count	网络异常后的重试次数，默认为 3
pause	重试时停顿的秒数，默认为 0

不过，get_hist_data()函数不能获取早于 2017 年 2 月 7 日（具体日期以实际命令执行结果为准）的数据。需要更早的数据时，可使用函数 get_k_data()。这两个函数的参数设置完全相同，明显不同的是 get_k_data()返回的数据按照日期先后排列。例如，获取代码为 000001 的股票自 2000 年 1 月 1 日至 2019 年 2 月 19 日的全部日线行情数据参数设置如下：

```
In [6]: df = ts.get_k_data(code='000001', start='2000-01-01',
                end='2019-02-19', ktype='D')
In [7]: df.head()
Out[7]:
         date       open   close   high     low      volume    code
0     2000-01-04   3.859   4.034   4.091   3.793    82160.86  000001
1     2000-01-05   4.047   3.983   4.157   3.970    93993.15  000001
2     2000-01-06   3.974   4.142   4.201   3.915   120222.09  000001
3     2000-01-07   4.190   4.309   4.360   4.168   229346.33  000001
4     2000-01-10   4.365   4.442   4.517   4.360   185210.78  000001
```

当然，还可以用其他函数获取更早日期的股票交易数据。例如：

```
In [8]: df = ts.bar('000001', conn = ts.get_apis(), freq = 'D',
                start_date = '1996-01-01', end_date = '')
```

这个 bar()接口函数是 Tushare 包提供的一个通用接口函数，它可以获取自股票上市以来任何起始日期的日线数据。本章对接口函数 bar()的使用不做进一步的说明，对此感兴趣的读者可以登录 Tushare 平台了解这个接口函数更为详细的使用方法。

除上述 3 个函数外，Tushare 包还提供一些 pro 接口函数返回股票历史行情数据。平台注册用户可以利用 pro 接口函数获取股票行情数据，但绝大部分 pro 接口函数要求用户具有一定的积分才可以调用（请参阅 Tushare 平台网页关于用户获取积分的操作指引）。例如，获取股票日线行情的 pro 接口函数的操作方法如下：

```
In [9]: pro = ts.pro_api()
In [10]: df = pro.daily(ts_code = '600008.SH', start_date = '20000501',
                        end_date = '20190808')
```

这个 pro 接口函数 pro.daily()的调用表示获取代码为 600008 的股票在给定起止日期内（返回数据的最早日期不会早于股票的首次上市日期）的股票交易数据（可能会因 Tushare 平台调整数据源而影响股票早期数据的获取）。注册用户目前使用这个接口函数没有积分要求，但若 1 分钟之内调用它的次数超过 200 次，则要求用户具备 5000 积分。

利用 pro 接口函数 pro.daily()获取的行情数据与利用 get_hist_data()函数获取的行情数据在结构上有些差别。函数 pro.daily()的返回数据采用默认索引，交易日期作为一个字段（列）

数据项，而 get_hist_data()函数的返回数据以交易日期作为索引。利用 head 方法可以查看返回值的结构：

```
In [11]: df.head()
Out[11]:
   ts_code   trade_date open  high  ...  change  pct_chg     vol     amount
0 600008.SH   20190808  3.28  3.32  ...   -0.01  -0.3040  118205.34  38886.705
1 600008.SH   20190807  3.31  3.33  ...   -0.01  -0.3030   97863.20  32318.541
2 600008.SH   20190806  3.31  3.34  ...   -0.09  -2.6549  220259.84  72588.431
3 600008.SH   20190805  3.40  3.41  ...   -0.03  -0.8772  103872.42  35213.327
4 600008.SH   20190802  3.36  3.42  ...    0.00   0.0000  155621.91  52594.915
[5 rows x 11 columns]
```

鉴于不同接口函数返回数据的结构存在或多或少的差别，因此在设计处理这些数据的程序时必须根据返回数据的结构选择合适的处理方法，或者对返回的数据项进行必要的修改，以满足其他数据处理语句的格式要求。

类似地，还有用来获取以周、月为周期的股票交易数据的 pro 接口函数。常用的 pro 接口函数如表 10.5 所示。

表 10.5　常用的 pro 接口函数

接口名称	功能说明	接口名称	功能说明
pro.daily()	A 股日线行情	daily_basic()	全部股票每日重要的基本面指标
pro.weekly()	A 股周线行情	pro.income()	上市公司财务利润表数据
pro.monthly()	A 股月线数据	pro.balancesheet()	上市公司资产负债表

除提供股票的历史数据外，Tushare 还提供股票交易行情的实时数据，即当天正在交易的股票价格数据。例如，通过 get_realtime_quotes()函数获取的股票交易实时分笔数据可以包括股票当前时刻报价列表和成交价格等信息、五档买入报价和五档卖出价格等数据项，共有 30 余项信息。操作过程如下：

```
In [12]: df = ts.get_realtime_quotes('300274') # 单个股票实时行情
In [13]: df[['code','name','price','bid','ask','volume','amount','time']]
         #需要显示的属性信息
```

如果需要一次性获取多只股票的实时分笔数据，那么可以将这些股票的代码表示成一个列表，作为函数的参数（最好不要超过 30 只股票）。例如：

```
In [14]: df = ts.get_realtime_quotes(['600460','000762','000725'])
```

还可以利用这个函数获取沪深两个市场的各类指数数据，这时需要用指数的字符代号作为函数的参数。例如，ts.get_realtime_quotes('sh') 表示获取 A 股上证指数。当然，也可以一次性获取多项指数的实时数据。例如：

```
In [15]: df = ts.get_realtime_quotes(['sh', 'sz', 'hs300', 'sz50', 'zxb', 'cyb'])
```

这样返回的数据同时包括上证指数、深圳成指、沪深 300 指数、上证 50、中小板及创业板等板块的实时价格信息。

总之，用户可以根据实际需要利用 Tushare 包的内建接口函数获取沪深两市股票不同周期（日、周、月）的（历史或实时）交易数据，或者获取上海证券交易所和深圳证券交易所提供的各类指数数据。

10.1.3　上市公司基本面数据

Tushare 平台提供的上市公司基本面数据包括财务状况、盈利状况、市场占有率、经营管理体制、

人才构成等各个方面的数据。除股票价格行情数据外，金融分析人员通常需要通过上市公司的基本面数据了解公司的投资价值。Tushare 提供上市公司基本面数据的接口函数如表 10.6 所示。

表 10.6 基本面数据的接口函数

接口函数名称	功能描述	接口函数名称	功能描述
get_stock_basics()	沪深股票列表	get_growth_data(2018,4)	成长能力
get_report_data(2018,3)	公司业绩报告	get_debtpaying_data(2018,4)	偿债能力
get_profit_data(2018,3)	公司盈利能力	get_cashflow_data(2018,4)	现金流量
get_operation_data(2018,3)	公司运营能力		

例如，使用函数 get_stock_basics()可以获取沪深两个市场的股票基本信息，这个接口函数的返回值是两个市场的股票基本信息：

```
In [1]: import  tushare  as  ts
In [2]: df = ts. get_stock_basics()
In [3]: df.info()
<class 'pandas.core.frame.DataFrame'>
Index: 3676 entries, 603662 to 603115
Data columns (total 22 columns):
name               3676 non-null object
industry           3676 non-null object
area               3676 non-null object
pe                 3676 non-null float64
outstanding        3676 non-null float64
totals             3676 non-null float64
totalAssets        3676 non-null float64
liquidAssets       3676 non-null float64
fixedAssets        3676 non-null float64
reserved           3676 non-null float64
reservedPerShare   3676 non-null float64
esp                3676 non-null float64
bvps               3676 non-null float64
pb                 3676 non-null float64
timeToMarket       3676 non-null int64
undp               3676 non-null float64
perundp            3676 non-null float64
rev                3676 non-null float64
profit             3676 non-null float64
gpr                3676 non-null float64
npr                3676 non-null float64
holders            3676 non-null float64
dtypes: float64(18), int64(1), object(3)
memory usage: 660.5+ KB
```

这些信息说明当前沪深两个市场共有 3676 只股票的基本信息。如果要查看某只股票的基本信息，那么可以从变量 df 中提取。df 是以股票代码作为索引的 DataFrame 结构。要查看股票代码为 600030 的个股信息，可以执行如下操作：

```
In [4]: df.loc['600030']
Out[4]:
name                中信证券
industry              证券
area                  深圳
pe                   23.98
outstanding          98.15
totals              121.17
totalAssets      6.14146e+07
liquidAssets             0
fixedAssets         778785
reserved         5.44224e+06
reservedPerShare        4.49
esp                    0.604
bvps                   12.64
pb                      1.53
timeToMarket        20030106
undp             5.51119e+06
perundp                 4.55
rev                    -4.53
profit                 -7.73
gpr                        0
npr                    26.88
holders               553777
Name: 600030, dtype: object
```

10.1.4　股票指数数据

股票指数是由证券交易所或金融服务机构编制的反映某一组（类）股票价格变动的一种股票综合价格数值。投资者容易了解具体某只股票的价格变化，但要逐一了解多只股票的价格变化则可能不胜其烦。为了反映多只股票价格变动的整体情况，证券交易所及一些金融服务机构已经编制并公开发布了数十个股票价格指数，股票投资者也习惯以股票指数作为考察股票市场价格变动的观察指标。如表 10.7 列出的 Tushare 平台 pro 版接口函数 index_basic()的输入参数和表 10.8 列出的输出参数信息所示，index_basic()函数可以获取我国当前已经发布的全部指数数据。

表 10.7　index_basic()的输入参数

参数名称	类　型	必　选	描　述
market	str	Y	交易所或服务商
publisher	str	N	发布商
category	str	N	指数类别

表 10.8　index_basic()的输出参数

参数名称	类　型	描　述	参数名称	类　型	描　述
ts_code	str	TS 代码	category	str	指数类别
name	str	公司简称	base_date	str	基期
market	str	交易市场	base_point	float	基点
publisher	str	发布商	list_date	str	发布日期

我国目前发布的 A 股指数的交易所或服务商信息如表 10.9 所示。index_basic()接口输入参数的 market 只能使用表 10.9 中所列的代码。

表 10.9 指数的市场代码

市场代码	说 明	市场代码	说 明
MSCI	MSCI 指数	CICC	中金所指数
CSI	中证指数	SW	申万指数
SSE	上交所指数	CNI	国证指数
SZSE	深交所指数	OTH	其他指数

例如，若需要获取上交所发布的各项股票指数，则可按照如下方式操作。

```
In [1]: import  tushare  as  ts
In [2]: pro = ts.pro_api()
In [3]: df = pro.index_basic(market = 'SSE')
In [4]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 606 entries, 0 to 605
Data columns (total 8 columns):
ts_code       606 non-null object
name          606 non-null object
market        606 non-null object
publisher     606 non-null object
category      606 non-null object
base_date     606 non-null object
base_point    606 non-null float64
list_date     605 non-null object
dtypes: float64(1), object(7)
memory usage: 38.0+ KB
In [5]: df.head(3)
Out[5]:
     ts_code   name   market publisher category  base_date  base_point list_date
0  000001.SH 上证综指  SSE  上交所    综合指数   19901219  100.00      19910715
1  000002.SH 上证 A 指  SSE  上交所    综合指数   19901219  100.00      19920221
2  000003.SH 上证 B 指  SSE  上交所    综合指数   19920221  100.00      19920221
```

虽然 Tushare 平台还提供了指数日线行情、指数成分和权重以及大盘指数每日指标等多个实用的 pro 接口函数，但它们都要求用户具备一定数量的积分才能正常获取数据。表 10.10 中的接口函数可以不要求用户的积分数量就能获取指数数据，但其返回数据的结构与 pro 接口函数返回数据的结构明显不同，因此在对函数返回数据做进一步处理时，需要考虑它们之间的差别。

表 10.10 获取市场指数的接口函数

函数调用格式	功能描述
get_hist_data('sh')	获取上证指数 k 线数据，其他参数与个股一致，下同
get_hist_data('sz')	获取深圳成指 k 线数据
get_hist_data('hs300')	获取沪深 300 指数 k 线数据
get_hist_data('sz50')	获取上证 50 指数 k 线数据
get_hist_data('zxb')	获取中小板指数 k 线数据
get_hist_data('cyb')	获取创业板指数 k 线数据

例如，执行函数调用命令 get_hist_data('sz50',start='2017-01-01',end='2019-02-13') 将返回上证 50 指数自 2017 年 1 月 1 日至 2019 年 2 月 13 日的日线行情数据。但这个接口函数能获取的数据起始日期不可早于 2017 年 3 月 24 日（这个日期并不固定，请以实际操作结果为准）。若需要获取更早的数据，则需要选用 get_k_data()接口函数，例如：

```
In [6]: df = ts.get_k_data('sz50',start = '2005-01-01',end = '2019-02-13', ktype = 'W')
In [7]: df.head()
Out[7]:
        date    open    close    high    low     volume     code
0  2005-01-07  836.99  823.62  836.99  819.44  12872218.0  sz50
1  2005-01-14  823.77  829.04  845.55  821.00  12286038.0  sz50
2  2005-01-21  821.19  831.85  835.24  798.35  21326228.0  sz50
3  2005-01-28  847.07  820.69  851.24  816.54  18992938.0  sz50
4  2005-02-04  817.93  872.88  877.39  810.72  27546577.0  sz50
```

显示的结果表示已经获取上证 50 指数周线行情数据。采用类似的方法可以获取各指数的日线行情、月线行情等数据。

10.2　股票行情数据的可视化

财经数据通常是反映一段较长时间内某个经济项目的时间序列数据。序列中的数据形式和结构可以多种多样，数据分析人员通常不容易了解序列数据变化的整体特征。一种较好的解决方法是将时间序列数据绘制成图表，选择一种比较直观的图表形象地反映各项数据在某段时间内的变化情况，即采用可视化方法展示序列数据的变化特征。

10.2.1　绘制股票 k 线图

不同种类财经数据的结构与变化特征是有差别的，适合描述不同种类数据特征的图表形式自然也会不同。线图和点图是金融分析者最常用的二维图，因为二维线图与点图比较容易展示金融数据的变化特征，并且绘制方法也较简单。本节以绘制股票 k 线图（也称蜡烛图）为例介绍 Python 语言实现时间序列数据可视化的基本方法。

在 Python 运行环境下利用 Tushare 平台数据绘制股票 k 线图的过程主要包含如下三个步骤：

第 1 步：确定数据来源。本节选择从 Tushare 平台获取股票行情数据，并利用 Tushare 包的内建接口函数获取股票行情数据，所以首先要使用命令 import tushare 导入 Tushare 包。

第 2 步：确定可视化的形式和实现工具。本节选择金融学图表包 mpl_finance 作为绘制股票价格 k 线图的工具，所以需要使用命令“from mpl_finance import candlestick_ochl”导入 mpl_finance 包中绘制 k 线图的子模块 candlestick_ochl。

mpl_finance 包是从 Matplotlib 中独立出来的图形包（执行“pip install mpl_finance”命令完成包的安装），通常用于绘制股票价格 k 线图及线图。由于 Tushare 包提供的 pro 接口函数与普通接口函数返回的数据结构存在差异，所以在调用绘图函数时需要确保数据格式的匹配。本节的程序中利用 pro 接口的 pro.daily()函数获取股票日线数据，并根据 mpl_finance 包的 candlestick_ochl()函数的参数格式要求对行情数据结构进行适当的调整。此外，输出图表中绘制的 k 线数量不宜太多，因为 k 线数量太多势必导致 k 线之间距离太小而使得 k 线图不够清晰。

第 3 步：确定绘制图表的输出工具。本节选择图表绘制包 Matplotlib 作为 k 线图的输出工具，因为 Matplotlib 包提供了丰富的图表输出功能，可以比较容易地对图表的结构布局、颜色及坐标

轴格式等诸多方面进行设置，使得图表更美观，也更容易理解。所以在程序中需要导入模块 Matplotlib，命令为“import matplotlib”。

如下程序运行输出的 k 线图如图 10.1 所示。程序中给出了少量注释，并且在程序的头部依次导入了程序需要的所有包。

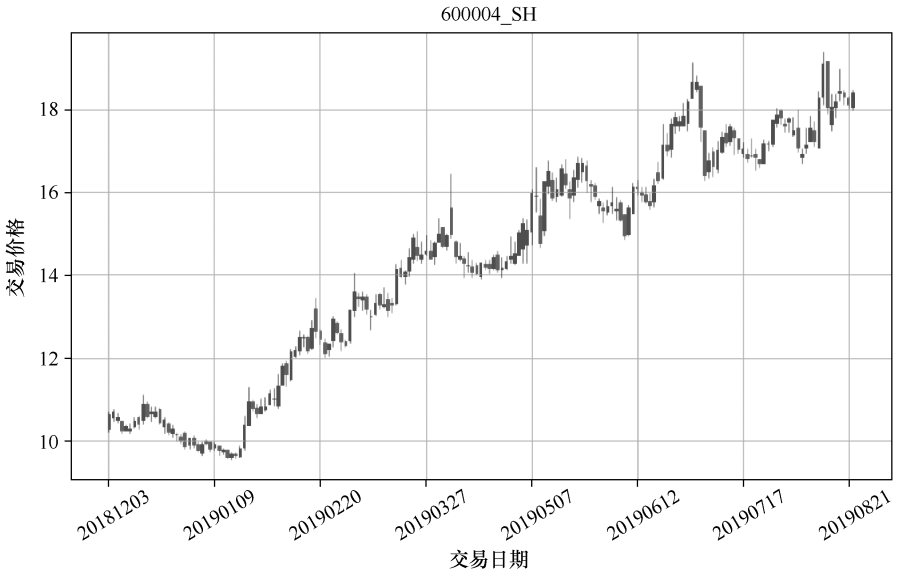


图 10.1 股票 k 线图

```
import tushare as ts
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import ticker
from matplotlib.pylab import date2num
from mpl_finance import candlestick_ochl
plt.rcParams['font.sans-serif'] = ['SimHei']   # 用来正常显示中文标签
pro = ts.pro_api()
code = '600004.SH'
df = pro.daily(ts_code=code, start_date='20181201')
df = df.sort_values(by='trade_date', ascending=True)  # 原始数据按照日期降序排列
df['trade_date2'] = df['trade_date'].copy()
df['trade_date'] = pd.to_datetime(df['trade_date']).map(date2num)
df['dates'] = np.arange(0, len(df))
def format_date(x, pos):
    if (x < 0) or (x > len(date_tickers)-1):
        return ''
    return date_tickers[int(x)]
date_tickers = df['trade_date2'].values
df2 = df.query('trade_date2 >= "20171001"').reset_index()
df2['dates'] = np.arange(0, len(df2))
date_tickers = df2['trade_date2'].values
fig, ax = plt.subplots(figsize=(12, 9))
```

```
    fig.subplots_adjust(bottom=0.2) # 控制子图
    candlestick_ochl(ax, quotes=df2[['dates', 'open', 'close', 'high', 'low']].values,
                     width=0.55, colorup='r', colordown='g', alpha=0.95)
    ax.xaxis.set_major_formatter(ticker.FuncFormatter(format_date))
    ax.set_ylabel('交易价格')
    plt.title(code)
    plt.grid(True)  # 添加网格，可有可无，只是让图像好看一些
    plt.xticks(rotation=30)  # 设置日期刻度旋转的角度
    plt.xlabel('交易日期')
    plt.show()
```

程序中对接口函数返回数据的日期进行了一些处理，目的是满足 candlestick_ohlc()绘制图形所需的参数格式，同时消除因股票没有发生交易的日期（即公司停牌日）在图中出现 k 线缺失而形成断点空白。要特别注意 pro.daily()函数返回数据的排列及其数据列的名称，因为这些信息直接影响程序语句的实现。绘制 k 线的方法有多种，请读者尝试其他的实现方法（创建一个完整的程序实现从数据获取到图表输出的全过程）。

10.2.2　绘制股票价格移动均线与成交量

股票价格是股票市场中买卖双方达成的成交价格。我国股票市场（上海证券交易所和深圳证券交易所）正常交易日从上午 9 点 15 分开始到 11 点 30 分结束，下午从 1 点 0 分开始到 3 点 0 分结束。每只正常交易的股票在一天的交易过程中会出现不同的交易价格，这些价格信息通常采用股价分时图和交易明细两种方式展示，但交易者很难从分时图展示的价格变动来预估未来数日股票价格涨跌的倾向。相对分时图而言，股票上午的开盘价格、下午的收盘价格、一天交易中出现的最高价格及最低价格等价格信息，比较容易体现连续多个交易日的股票价格变化走势。

市场交易经验表明，股票的成交量变化对股票价格的变化趋势具有一定的影响。股票成交量是指当日买卖成交的股票数量（单位：手），它是反映股票活跃水平的重要指标，成交量越大，表明该股票越活跃、越受股票交易者关注。因此，很多股票投资者喜欢将成交量变化作为参与股票交易的重要指标。

为了便于直观地了解股票价格及其成交量在一定时间范围内的变化趋势，可以绘制一张图表统一描述股票价格与成交量数据变化的情况。绘制这类股票行情数据组合图形的方式有多种，读者可以根据自己的兴趣尝试不同的实现方式，仔细比较这些方式彼此的优劣。创建如下程序可以绘制股票的日线行情 k 线、收盘价格移动平均线及日成交量的二维图形，如图 10.2 所示。

```
    import  tushare  as  ts
    import  pandas  as  pd
    import  numpy  as  np
    import  matplotlib.pyplot  as  plt
    from  matplotlib  import  ticker
    from  matplotlib.pylab  import  date2num
    from  mpl_finance  import  candlestick_ochl

    plt.rcParams['font.sans-serif']=['SimHei']   #用来正常显示中文标签
    pro = ts.pro_api()
    code='600519.sh'
    df = pro.daily(ts_code=code, start_date='20180801')
    df = df.sort_values(by='trade_date', ascending=True) #原始数据按照日期降序排列
```

```
df['trade_date2'] = df['trade_date'].copy()
df['trade_date'] = pd.to_datetime(df['trade_date']).map(date2num)
df['dates'] = np.arange(0, len(df))
#计算移动平均值
df["10"] = np.round(df["close"].rolling(window = 10, center = False).mean(), 2)
df["30"] = np.round(df["close"].rolling(window = 30, center = False).mean(), 2)
df["60"] = np.round(df["close"].rolling(window = 60, center = False).mean(), 2)
df["120"] = np.round(df["close"].rolling(window = 132, center = False).mean(), 2)

def format_date(x,pos):  #设置坐标格式
    if (x<0) or (x>len(date_tickers)-1):
        return ''
    return date_tickers[int(x)]

date_tickers = df['trade_date2'].values
df2 = df.query('trade_date2 >= "20180601"').reset_index()
df2['dates'] = np.arange(0, len(df2))
date_tickers = df2['trade_date2'].values
from matplotlib.gridspec import GridSpec  # 用于控制子图
figure = plt.figure(figsize=(10, 6))
gs = GridSpec(3, 1)
ax1 = plt.subplot(gs[:2, :])
ax2 = plt.subplot(gs[2, :])
candlestick_ochl(ax=ax1,
                 quotes=df2[['dates', 'open', 'close', 'high', 'low']].values,
                 width=0.55,colorup='r',colordown='g',alpha=0.75)  # 绘制 K 线图
xmajorLocator = ticker.MultipleLocator(22) #将 x 轴主刻度设置为 22 的倍数
ax1.xaxis.set_major_locator(xmajorLocator)
ax1.xaxis.set_major_formatter(ticker.FuncFormatter(format_date))
plt.grid(True)   #添加网格，可有可无，只是让图像好看一些
for ma in ['10', '30','60', '120']:  # 绘制均线
    ax1.plot(df2['dates'], df2[ma])
ax1.legend(('sma10','sma30','sma60','smz120'))  #显示图例)
ax1.set_title(code, fontsize=20)
ax1.set_ylabel('交易价格')
ax2.xaxis.set_major_formatter(ticker.FuncFormatter(format_date))
for index, row in df2.iterrows():
    if(row['close'] >= row['open']):
        ax2.bar(row['dates'],row['vol'],width = 0.55,color='red') #/1000000
    else:
        ax2.bar(row['dates'],row['vol'],width = 0.55,color='green')
ax2.set_ylabel("成交量（单位：手）") #设置 y 轴标题
ax2.set_ylim(0,df2['vol'].max()*1.2) #设置 y 轴范围/100000000
ax2.xaxis.set_major_locator(xmajorLocator)
plt.xticks(rotation=30)              # 设置日期刻度旋转的角度
plt.show()
```

由于 k 线图和成交量图都是有色彩标记的矩形，所以在绘图前首先要考虑数据可视化的清晰要求。如果绘图空间不足以展示预期的数据规模（与时间段长度成正比），那么就要扩大绘图区

域大小或降低数据规模。如果要在同一个绘图板上绘制多个子图，那么要仔细调整子图之间的数据匹配和坐标刻度数据的匹配，避免日期、数值（价格、成交量）与图形出现错配。图 10.2 展示了股票 k 线、收盘价格的 4 条移动平均线及每个交易日的成交量。

在设计程序时要注意 Tushare 接口函数返回的数据结构。因为有些接口函数返回的数据本来就包含价格的多个时间跨度移动平均值序列和/或成交量移动平均值序列（如 5 日、10 日、20 日移动平均值），而有些接口函数返回的数据并不包含移动平均值序列。因为本程序使用的`pro.daily()`函数没有返回移动平均值，所以需要计算平均值序列。

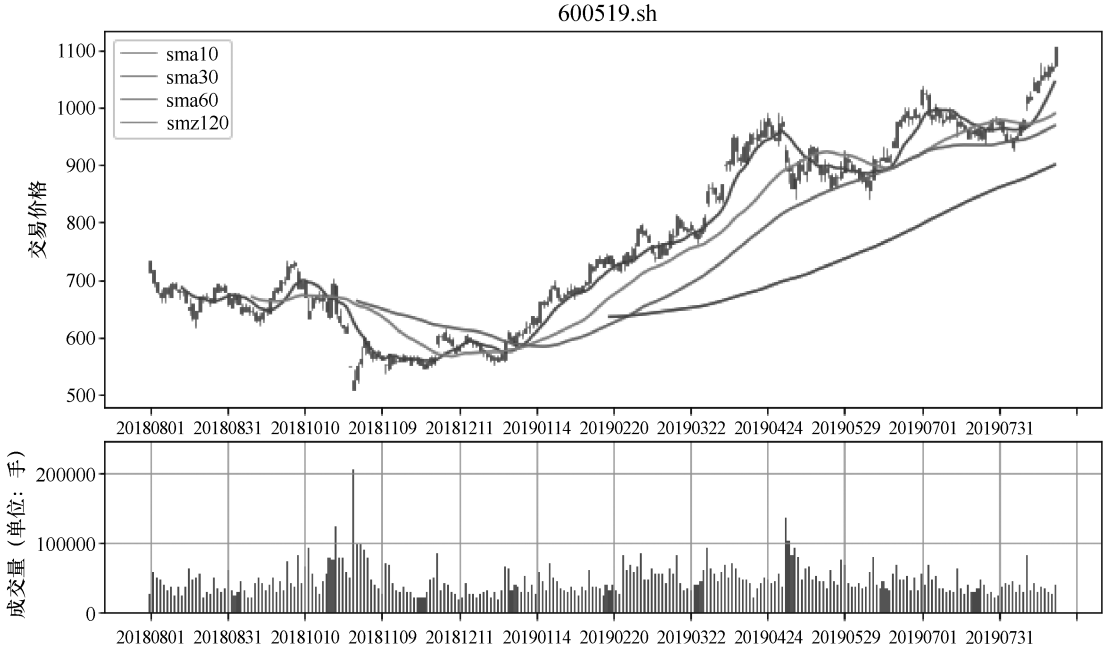


图 10.2　股票 k 线、移动平均线、成交量

除日线外，Tushare 还提供股票价格周线和月线数据。读者可以根据自己的需要决定获取日线、周线或月线数据，并选择合适的接口函数及其参数。

10.3　优质基本面的股票池创建

上市公司的基本面数据是体现公司经营历史业绩的重要凭据，也是投资者判断公司未来发展前景的重要依据，金融分析人员和股票投资者需要通过分析公司基本面质量来评估公司股票的投资价值。从 Tushare 平台获取的上市公司基本面数据主要包括定期公布的公司经营业绩报告、盈利能力、营运能力、成长能力、偿债能力及现金流等数据，这些数据分别体现公司不同层面的经营状况。

我国股票市场（上海证券交易所和深圳证券交易所）已有近 3600 只正常交易的股票，普通投资者选择理想的投资标的时，不可能有足够时间和精力分析全部上市公司的基本面数据。因此，投资者有必要利用一些具有经验指导意义的关键指标对市场中的股票进行自动化筛选，即依靠计算机程序从全部上市公司股票中筛选出优质基本面的股票，从而显著提高投资者寻找优质公司股票的工作效率。例如，从公司的盈利能力、成长能力和现金流量等多个类别基本面数据中的部分

指标项筛选出优质股票作为候选投资标的。

筛选优质股票的基本步骤为如下 4 步：

① 利用 Tushare 包内建的基本面数据接口函数获取全部股票的基本面数据。

② 根据经验确定反映基本面质量的关键指标项，从全部股票的基本面数据中提取关键指标项对应的数据序列。

③ 利用 Pandas 内建函数将多个基本面数据序列合并成一个 DataFrame 数据。

④ 根据关键指标项的重要程度确定数据序列的排序参数，对合并的数据进行排序。从排序结果中选择各项指标综合排序靠前的若干行，这些行对应的股票便是基本面质量相对较高的股票集合（即优质股票池）。

表 10.11 列出了获取公司盈利能力、成长能力和现金流基本面数据的接口函数及其返回参数的信息。

表 10.11 基本面数据的接口函数及其返回参数信息

接口函数名称	返回参数	功能说明
get_profit_data()	code，代码；name，名称；roe，净资产收益率（%） net_profit_ratio，净利率（%）；gross_profit_rate；毛利率（%）；net_profits，净利润(万元)；esp，每股收益 business_income，营业收入(百万元)；bips，每股主营业务收入(元)	按年度、季度获取盈利能力数据
get_growth_data()	code，代码；name，名称；mbrg，主营业务收入增长率（%）；nprg，净利润增长率（%）；nav，净资产增长率；targ，总资产增长率；epsg，每股收益增长率；seg，股东权益增长率	按年度、季度获取成长能力数据
get_cashflow_data()	code，代码；name，名称；cf_sales，经营现金净流量对销售收入比率；rateofreturn，资产的经营现金流量回报率；cf_nm，经营现金净流量与净利润的比率；cf_liabilities，经营现金净流量对负债比率；cashflowratio，现金流量比率	按年度、季度获取现金流量数据

从表 10.11 中列出的函数的返回数据项目来看，接口函数返回的数据种类非常丰富，可用来体现公司投资价值的数据项有很多组合。例如，从表 10.11 所列三类数据的返回参数中选择净利率、净资产收益率、每股收益增长率、净利润增长率、每股收益增长率、经营现金净流量与净利润的比率及现金流量比率等 7 项作为关键评价指标，综合利用这 7 项指标的值反映公司的基本面质量。通常情况下，这些指标值越大，表示公司的基本面质量越高，公司股票的投资价值越大。如下程序演示了一种利用财务指标数据筛选优质公司股票的方法。

```
import tushare as ts
import pandas as pd
import datetime

this_year = datetime.datetime.today().year
this_month = datetime.datetime.today().month
if this_month >= 11:  # 今年的三季报已经发布
    fin_year = this_year
    fin_sea = 3
elif this_month >= 5:  # 通常要在 4 月底才发布完上年度年报，当然也可选今年一季报
    fin_year = this_year-1
    fin_sea = 4
```

```
    else:
        fin_year = this_year-1
        fin_sea = 3
    df1 = ts.get_profit_data(fin_year, fin_sea)
    df2 = ts.get_growth_data(fin_year, fin_sea)
    df3 = ts.get_cashflow_data(fin_year, fin_sea)
    df_merge = pd.merge(df1[['code', 'net_profit_ratio', 'roe', 'eps']],
                        df2[['code', 'nprg', 'nav']], on='code', how='left')
    df_merge = pd.merge(df_merge, df3[['code', 'cf_nm', 'cashflowratio']],
                        on='code', how='left').dropna()
    focus_df = df_merge.sort_values(['nprg', 'net_profit_ratio', 'cf_nm', 'nav',
                                     'roe', 'eps', 'cashflowratio'], ascending=False)
    if focus_df.iloc[:, 0].size > 100:
        select_df = focus_df[['code', 'nprg', 'net_profit_ratio', 'cf_nm', 'nav',
                              'roe', 'eps', 'cashflowratio']].head(100)
    else:
        select_df = focus_df[['code', 'nprg', 'net_profit_ratio', 'cf_nm', 'nav',
                              'roe', 'eps', 'cashflowratio']]
    select_df.to_csv('focus'+str(fin_year)+str(fin_sea)+'.csv')
```

我国 A 股上市公司的财务报告披露时间规定：4 月 30 日前披露一季报，8 月 31 日前披露半年报，10 月 30 日前披露三季报，披露年报的时间是次年度的 4 月 30 日前。程序中利用 datatime 包的日期函数确定可获取的财报数据。

如果在运行程序前预先保存了相应的财务数据文件，那么可以直接从本地数据文件中读取财务数据，从而避免每次运行程序时都要连网下载相同数据的情形。关于对合并报表进行排序的关键列次序，有兴趣的读者可以多做一些次序调整，比较最终保留的数据表 select_df 中的股票集合及其排序情况，考察不同指标次序的排序操作结果有何不同，并找到对应的股票以了解股票价格近 3 年的价格变化情况。

从股票市场的现实来看，选择具有优质基本面的上市公司股票的关键指标并没有统一的标准，一方面是因为不同的财务指标反映公司的侧重点时存在差别，另一方面是因为不同行业公司的财务数据指标在数值上的可比性尚无定论。本例仅提供一种利用 Python 程序提高优质公司股票筛选效率的方法，而不提供一种筛选优质股票的投资依据。

10.4　本章小结

目前 Python 程序可从 Web 上获取各类财经数据，其中有些数据平台需要缴纳数据服务费才可以下载数据，但 Tushare 平台提供免费的数据服务。本机安装 Tushare 包后，Python 程序就可利用 Tushare 包的内建函数便捷地获取各类财经数据。本章主要讨论从 Tushare 平台获取股票数据的多个接口函数及其使用方法，给出了利用股票交易数据序列绘制股价 k 线图、移动平均线、成交量等数据的可视化 Python 程序，展示了一种利用 Python 程序筛选优质公司股票池的方法。

习题

1. 利用 Tushare 包提供的接口函数获取当前电影市场的票房信息，确定当前最受观众喜爱的电影名称及其票房收入。

2. 获取我国自 2008 年以来各年度的货币供应量数据，比较货币供应量 M2 的增长速度与我国 GDP 同期的增长速度（图表输出）。
3. 设计一个程序以股票价格的日线数据为基础分别绘制周 k 线和月 k 线图，要求在图形中不出现股票停牌交易日产生的 k 线位置空缺。
4. 设计一个程序获取上市公司的基本面数据，对不同行业的上市公司股票以若干基本面指标项排序，筛选出每个行业中基本面排名前 5 的上市公司股票名单。
5. 设计一个程序分别绘制上证综合指数和深证成份指数的 60 日、120 日及 250 日价格移动平均线。要求：如果行情数据涵盖的时间小于 120 日，那么仅绘制 k 线图；如果行情数据涵盖的时间大于 2 年，那么输出图形不绘制 k 线，但需要绘制收盘价移动均线图；否则，程序输出图形需要绘制 k 线、移动平均线及每日成交量。

第 11 章　Python 金融分析应用

金融数据分析是 Python 的重要应用领域之一。当前各版本的 Python 都拥有种类丰富且功能强大的金融计算工具包，这些工具包为金融数据分析人员开发 Python 应用程序提供了极大的便利。本章首先介绍 NumPy 包中关于分期投资与贷款的金融函数及其使用方法，然后以股票收益率和波动率的计算为例介绍金融时间序列数据分析的 Python 实现方法，接着介绍计算 MACD 和 KDJ 等股票价格技术分析指标值的 Python 实现方法，最后介绍应用 Python 分析国内生产总值等多个宏观经济数据变化的方法。

11.1　实用 NumPy 金融函数

当前的 NumPy 已然是一个涉及诸多应用领域的强大计算工具包，其提供的金融计算功能日益丰富，为金融数据分析程序开发人员提供了便捷的实用函数，极大地减轻了金融数据分析软件开发者的编程工作量。本节介绍几个与人们日常生活关系比较密切的 NumPy 金融函数（见表 11.1），建议读者也多了解其他实用的 NumPy 金融函数及其使用方法。

表 11.1　NumPy 的常用金融函数

函数名称	功能说明
fv(rate,nper,pmt,pv[,when])	计算金融资产在未来某一时间点的价值，或称终值、未来值
pv(rate,nper,pmt,fv[,when])	计算金融资产当前的价值，或称现值
pmt(rate,nper, pv, fv[,when])	计算每期需支付借款的金额
nper(rate,pmt,pv,fv[,when])	计算定期付款的期数

在 IPython 操作界面导入 NumPy 库后就可用这些 NumPy 函数执行相应的计算：

```
In [1]: import numpy as np
```

① 终值函数 fv(rate, nper, pmt, pv[,when])

终值函数 fv()以固定利率计算一项投资或贷款在未来某一时间点具有的价值（金额）。参数 rate 表示存款/贷款每期的利率（假定在终止时间点之前利率维持不变），nper 表示存款/贷款的总期数，pmt 表示存款/贷款每期支付的金额（包括本金和利息，但不包括其他费用或税款），pv 表示当前的存款/贷款总金额，参数 when 可选，when='begin'或 1 表示在每一期的期初进行付款，when='end'或 0 表示在期末付款，缺省时 when 参数为期末付款。

【例 1】王女士今天去银行办理了一项 2 年期的存款业务，且存入 10 万元人民币，并从下个月开始的每月存入 1 万元人民币。假设以年利率为 4%连续存 2 年，那么 2 年后（即最后一笔存款 1 万元人民币存满一月的最后一天）王女士的存款总额将是多少？

根据题设，这项存款计划在未来 2 年的利率保持 4%，先后存入 24 次 1 万元人民币，即存款期数为 24，且首期存款为 10 万元人民币。那么 2 年后（期末付款）可取的存款总额如下：

```
In [2]: np.fv(rate = 0.04/12, nper = 24, pmt = -10000, pv = -100000, when = 'end')
Out[2]: 357743.1733931053
In [3]:np.fv(rate = 0.04/12, nper = 24,pmt = -10000,pv = -100000)
```

```
Out[3]: 357743.1733931053
```

其中 pmt 和 pv 为负值时代表现金流出，首笔存款额 pv 不宜低于每期存入额 pmt，尽管 pv=0 时也可以正常计算，但与实际存贷款业务明显不符。余下各函数调用参数中皆省略 when 参数，意味着使用 when 的缺省值'end'。

如果不分多期存入本金（即 pmt=0），而只是一次性存入 10 万元人民币本金，那么当前存入的 10 万元人民币本金在 2 年后的终值如下：

```
In [4]:np.fv(rate = 0.04/12, nper = 24, pmt = 0, pv = -100000)
Out[4]: 108314.29591590662
```

【例 2】有些资产是每年贬值的，例如汽车、飞机、机床等资产存在自然贬值特征。现在假设花 20 万元人民币买了一辆新车，估计每年贬值 10%（即利率为-10%），那么 5 年后这辆汽车的价值（金额）如下：

```
In [5]: np.fv(rate = -0.1, nper = 5, pmt = 0, pv = -200000)
Out[5]: 118098.00000000001
```

从上述例子可以看出，fv(rate,nper,pmt,pv[,when])计算的是以规模为 pmt 的资金在给定利率 rate 的情况下重复累计 nper 次形成的总价值，而 pv 是初始资金价值。对于存款业务来说，pv 可以是 0，若 pv 是负数，则表示资金借出（即存入）。对于贷款业务来说，pv 是贷款方贷入现金额度，所以在参数中要设置为正数，且不能设置为 0。

函数 fv()的各项参数数据之间的关系如下：

```
fv + pv*(1+rate)**nper + pmt*(1 + rate*when)/rate*((1 + rate)**nper - 1) = 0
```

式中 when=0 或 1。从这个关系式也比较容易理解在现金流入情况下 pv 应该取正值，在现金流出情况下 pv 应该取负值。

② 现值函数 pv(rate, nper, pmt ,fv)

现值函数 pv()是指资产（未来价值金额）在当前时刻的价值（金额）。NumPy 中的 pv()函数需要利率、期数、每期支付金额和未来某时间点的预期金额（终值）作为参数，函数的返回结果为现值（即初始价值金额）。现值函数 pv()和终值函数 fv()是对称的。

【例 3】假设年利率为 10%，王女士希望 5 年后可以得到存款 50 万元人民币，并计划每月存 5000 元人民币，那么首笔存款金额应该是多少？也就是要计算开始这项存款计划之初应该存入银行多少钱才能达到预期的 50 万元人民币存款。用 pv()函数求解的参数设置如下：

```
In [6]: np.pv(rate = 0.1/12, nper = 60, pmt = -5000, fv = 500000)
Out[6]: -68567.4506156077
```

函数参数列表中的利率需要化成月利率，5 年共有 60 个月，也就需要分 60 次存入 5000 元人民币，预期存款总额就是这项存款计划可产生的终值。如果将这个计算结果-68567.45（舍入到分）应用到终值函数 fv()，即

```
In [7]: np.fv(rate = 0.1/12, nper = 60, pmt = -5000, pv = -68567.45)
Out[7]: 499999.9989871351
```

那么这个计算结果也体现了终值函数 fv()和现值函数 pv()的对称关系。

参数 pmt 也可以等于零。例如，要计算 5 年后的 10 万元人民币相当于现在的多少钱，可以表示如下：

```
In [8]: np.pv(rate = 0.02, nper = 5, pmt = 0, fv = 100000)
Out[8]: -90573.08098299158
```

其中 rate=0.02 是假设的年均通货膨胀率，即 2%。

③ 年金函数 pmt(rate, nper, pv, fv)

年金函数 pmt()计算以固定利率和等额本息方式每期支付给银行的款项（包含本金和利息）。函数的利率参数 rate 是根据年利率计算的期利率，若按月还款则为年利率除以 12，若按季度还款则为年利率除以 4。参数 nper 是整项贷款需要偿还的期（次）数，参数 pv 是从银行获得贷款的总金额。

【例 4】李先生计划从银行贷款 100 万元人民币买房子，贷款利率为 6.5%，若计划分 15 年还清全部贷款，那么每月需要还贷多少金额？

用 pmt()函数计算每月的按揭额度如下：

```
In [9]: np.pmt(rate = 0.065/12, nper = 15*12, fv = 0, pv = 1000000)
Out[9]: -8711.073652973655
```

④ 分期函数 nper(rate, pmt, pv, fv)

分期函数 nper()计算固定利率月等额本息还款方式定期付款的期数。函数的参数 rate 是存款/贷款每期的利率（根据年利率计算），pmt 是存款/贷款每期需要支付的金额，pv 是存款/贷款金额现值，fv 是存款/贷款终值（若是贷款，终值为 0；若是存款，终值为本息和）。

【例 5】张先生计划从银行贷款 100 万元人民币投资养殖，贷款利率为 6.5%，每月还贷 10000 元人民币，那么总共需要多久才能还清全部贷款？

因为目前银行通常要求客户按月还款，所以需要计算还款的总月份数：

```
In [10]: np.ceil(np.nper(rate = 0.065/12, pmt = -10000, pv = 1000000, fv = 0))
Out[10]: 145.0
```

这里利用 NumPy 的 ceil()函数对 nper()计算的结果进行取整得到整数值。

上述各例表明，我们可以利用这 4 个金融函数方便地计算日常生活中一些存款和贷款相关的数据。无论是投资还是贷款计划，利用这些金融函数都能简单方便地计算不同利率、期限、分期、额度等情况下的收益或成本。如下程序是计算商业房贷和公积金房贷的还款成本差异，其中计算按揭总款终值时使用的是目前一年期定期存款利率，在具体应用中需要调整为实际可供参考的利率。

```
import numpy as np
import matplotlib.pyplot as plt
dp_rate = 0.015  # 2015 年 10 月人民银行发布的 1 年期存款利率
rates = [0.0441, 0.0325]  # 5 年以上的商业房贷利率和公积金房贷利率
loan_pv = [50, 100, 150, 200]    # 单位：万元人民币
loan_nper = [5, 10, 15, 20]      # 单位：年
repay_pmt = np.zeros((len(rates), len(loan_pv), len(loan_nper)))   # 按揭月供金额
repay_fv = np.zeros((len(rates), len(loan_pv), len(loan_nper)))    # 实际还款未来终值
for r in range(len(rates)):
    for p in range(len(loan_pv)):
        for n in range(len(loan_nper)):
            repay_pmt[r][p][n] = round(np.pmt(rates[r]/12,
                                    loan_nper[n]*12, loan_pv[p])*10000, 2)
            repay_fv[r][p][n] = round(np.fv(dp_rate/12, loan_nper[n]*12,
                                     repay_pmt[r][p][n], 0), 2)
plt.rcParams['font.sans-serif'] = ['SimHei']   # 用来正常显示中文标签
fig, axes = plt.subplots(2, 2, figsize = (12, 7))
fig.suptitle('商业房贷与公积金房贷偿还终值比较')
axes[0, 0].plot(loan_nper, np.round(repay_fv[0][0][:]/10000, 2), label = '商业房贷')
axes[0, 0].plot(loan_nper, np.round(repay_fv[1][0][:]/10000, 2), label = '公积金房贷')
axes[0, 0].set(xticks = loan_nper, xlabel = '还款年限', ylabel = '50 万按揭终值')
```

```
        axes[0, 0].legend(loc = 0)
        axes[0, 1].plot(loan_nper, np.round(repay_fv[0][1][:]/10000, 2), label = '商业房贷')
        axes[0, 1].plot(loan_nper, np.round(repay_fv[1][1][:]/10000, 2), label = '公积金房贷')
        axes[0, 1].set(xticks = loan_nper, xlabel = '还款年限', ylabel = '100 万按揭终值')
        axes[0, 1].legend(loc = 'upper left')
        axes[1, 0].plot(loan_nper, np.round(repay_fv[0][2][:]/10000, 2), label = '商业房贷')
        axes[1, 0].plot(loan_nper, np.round(repay_fv[1][2][:]/10000, 2), label ='公积金房贷')
        axes[1, 0].set(xticks=loan_nper, xlabel = '还款年限', ylabel = '150 万按揭终值')
        axes[1, 0].legend(loc = 'upper left')
        axes[1, 1].plot(loan_nper, np.round(repay_fv[0][3][:]/10000, 2), label = '商业房贷')
        axes[1, 1].plot(loan_nper, np.round(repay_fv[1][3][:]/10000, 2), label = '公积金房贷')
        axes[1, 1].set(xticks = loan_nper, xlabel = '还款年限', ylabel = '200 万按揭终值')
        axes[1, 1].legend(loc = 'upper left')
        plt.show()
```

由两种贷款利率的差别导致的按揭款终值（按照 1 年期定期存款利率计算）差异如图 11.1 所示，图 11.1 中的两根线段分别表示还款总额随还款年限变化的趋势，表明还款总额随贷款额度和/或贷款年限的增加而显著增加。因此，在选择住房贷款时应该详细计算不同利率与不同贷款期限所产生的还款总额及各期还款金额等款项的差别，达到尽量减少自身经济压力的目的。

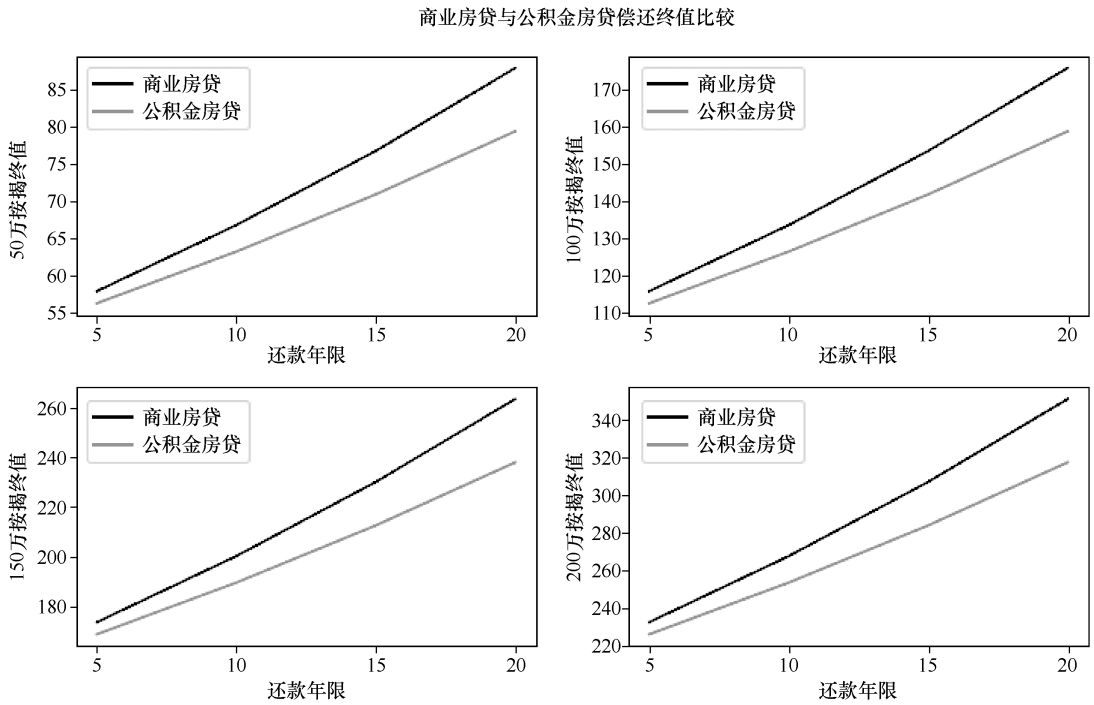


图 11.1　不同贷款利率下的还款额

由于商业贷款利率通常明显高于公积金贷款利率，所以同样还款期数的商业贷款按揭还款总额会远远高于公积金贷款按揭的还款总额。另一方面，因为按揭时间越长（还款期数越多）会导致偿还的资金总额越多（即终值越大），因此在资金允许时应该优先选择较短的贷款按揭年限。

11.2　股票的收益率和波动率

股票收益是指股票投资者以买卖股票方式所取得的收益（也称股利）。股票收益率是反映股票收益水平的指标，其值等于投资者卖出股票所获得的股票收益额与投资者购买该股票所支出金额的比值。影响股票收益率的因素主要包括股票价格涨跌、公司分配的股份红利、公司配送的新增股份、交易手续费及印花税等。本节讨论的收益率仅考虑股票购买者持有股票期间因股票价格涨跌产生的收益率，即投资者从买入股票到卖出股票所获得的差价与股票买入价的比率，且不考虑交易手续费等交易成本。

根据计算方法的不同，股票收益率分为简单收益率（也称百分比收益率、算术收益率）和对数收益率。简单收益率是指相邻两个价格之间的变化率，例如两个连续交易日的后一个交易日的收盘价和前一个交易日的收盘价之差与前一个交易日收盘价的比率。对数收益率是指相邻两个价格之比的对数值。由于在金融领域计算跨期复合收益率时要求收益率具有对称性和时序可加性等统计性质，所以更多情况下采用对数收益率表示股票收益率。

11.2.1　收益率计算

虽然利用 Tushare 接口函数获取的股票日线数据包含股票在各交易日的开盘价格、收盘价格、最低价格和最高价格等数据项，但本节计算股票投资收益率时仅以各交易日收盘价格作为交易价格（即投资者买入股票的价格是买入日的收盘价格，卖出股票的价格是卖出日的收盘价格），并假设投资者自买入日到卖出日一直持有相同数量的股票。

设投资者持有的某只股票在第 i 个交易日的收盘价为 P_i，在第 $i+1$ 个交易日的收盘价为 P_{i+1}，那么该只股票第 $i+1$ 个交易日相对第 i 个交易日的简单收益率为

$$r_s = (P_{i+1} - P_i) / P_i$$

相应的对数收益率为

$$r_e = \ln \frac{P_{i+1}}{P_i}$$

这两个收益率的最大差别如下：n 个交易日的累计对数收益率等于这 n 个交易日中每个交易日的对数收益率之和，但 n 个交易日的累计简单收益率并不等于这 n 个交易日中每个交易日的简单收益率之和。这是因为简单收益率的变化具有不对称性，即股票价格上涨与下降的数值相同而其简单收益率却不对称。例如，假设某只股票在第 i 个交易日的收盘价格是 10 元人民币，在第 $i+1$ 个交易日的收盘价格为 11 元人民币，而在第 $i+2$ 个交易日的收盘价格是 10 元人民币。股票价格虽然经过两个交易日后依然是 10 元人民币，但在第 $i+1$ 个交易日的简单收益率是 10%，而在第 $i+2$ 个交易日的简单收益率是−9.1%，所以这两个收益率之和并不等于零，但对应这两个交易日的对数收益率之和等于零。实际上，n 个交易日的累计简单收益率等于这 n 个交易日中每个交易日的简单收益率加 1 后的乘积再减 1。

本节的实例程序利用 Tushare 的内建函数 `get_k_data()`来获取股票的日线数据，该函数返回的 `DataFrame` 数据包含 `date`、`open`、`close`、`high`、`low`、`volume` 和 `code` 等列，分别表示交易日期、开盘价格、收盘价格、最高价格、最低价格、成交量和股票代码，在计算收益率时直接使用它的 `close` 数据列。

根据需要计算收益率的周期（日、周、月、季、年）确定获取合适周期的行情数据，尽管以

日线行情数据也可以计算周、月、季度或年等时间周期的收益率，但如果可以获取股票的同周期的行情数据，那与采用日线行情数据来计算周（月、季、年）收益率相比可以明显减少计算量。如下程序计算股票的日收益率，所以采用股票的日线行情数据分别计算指定时间内各交易日的简单收益率与对数收益率。

```
import tushare as ts
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import ticker

startday = '2018-01-01'
endday = '2019-08-07'
tscode = '600519'
df = ts.get_k_data(tscode, start = startday, end = endday)
df['simp_ret'] = np.round( (df['close']-df['close'].shift(1))/df['close'].shift(1), 4)
df['log_ret'] = np.round(np.log(df['close']/df['close'].shift(1)), 4)
df['simp_ret_cum'] = (df['simp_ret']+1).cumprod()-1
df['log_ret_cum'] = df['log_ret'].cumsum()
df = df.dropna()  # 消除第一行无效数据
data = {'date': pd.Series(df['date'].values),
        'close': pd.Series(df['close'].values),
        'simp_ret_cum': pd.Series(df['simp_ret_cum'].values),
        'log_ret_cum': pd.Series(df['log_ret_cum'].values)
        }
df2 = pd.DataFrame(data)  # 重新编排索引，有利于设置坐标刻度
plt.rcParams['font.sans-serif'] = ['SimHei']  # 用来正常显示中文标签

def format_date(x, pos):
    if x < 0 or x > len(date_tickers)-1:
        return('')
    return(date_tickers[int(x)])

date_tickers = df2['date'].values  # 生成横轴的刻度名字
fig, ax = plt.subplots(figsize = (16, 9))
fig.subplots_adjust(bottom = 0.2)
ax.set_xlim([0, len(date_tickers)+12])
ax.xaxis.set_major_formatter(ticker.FuncFormatter(format_date))
ax.xaxis.set_major_locator(ticker.LinearLocator(22))
plt.setp(plt.gca().get_xticklabels(), rotation = 30, horizontalalignment = 'right')
ax.plot(df2['simp_ret_cum']*100, label = '简单收益率')
ax.plot(df2['log_ret_cum']*100, label = '对数收益率')
ax.legend(loc = 'upper left')
plt.title(tscode)
plt.xlabel('交易日期')
plt.ylabel('收益率 %')
ax.grid(True)
plt.show()
```

为便于比较两种收益率的差别，这里绘制了同一时期内两种累计收益率的变化情况，如图 11.2 所示。从图中的累计收益率曲线可以看出，简单收益率的累计收益率和对数收益率的累计收益率整体走势是相近的，但在振荡频率较高的情况下，这两种累计收益率数值可能会出现明显差异，主要是因为简单收益率不具有对称性，而且累计简单收益率 r_s 与累计对数收益率 r_e 满足关系式 $r_e = \ln(r_s + 1)$ 。

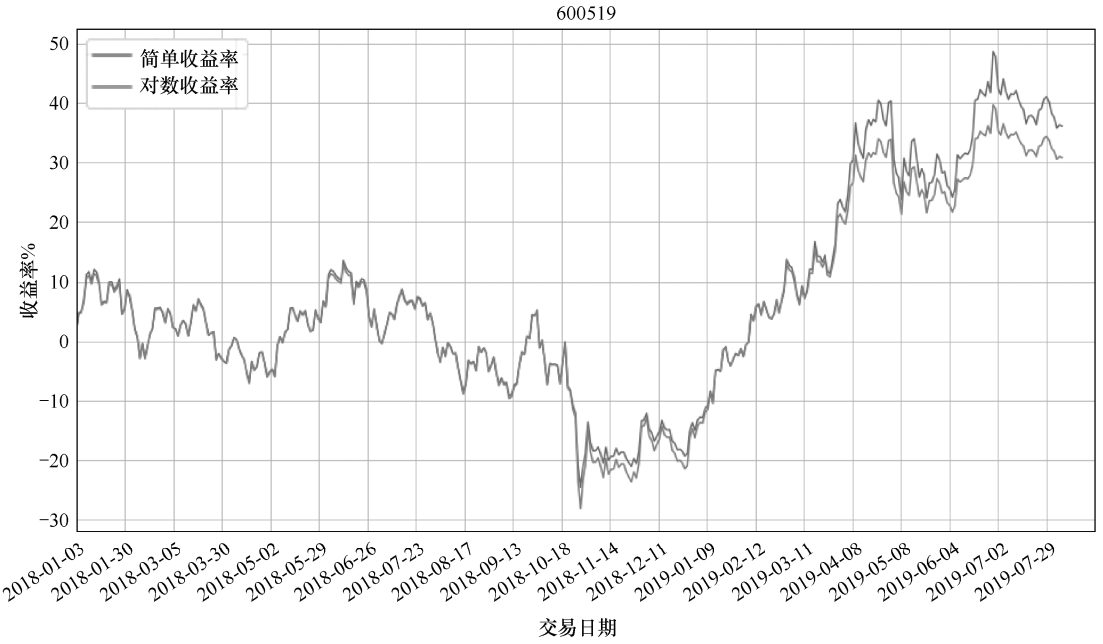


图 11.2　股票累计收益率

程序中如果利用 Tushare 的 pro.daily()接口函数获取日线行情数据，因为 pro.daily()函数的返回数据已经包含日涨跌幅数据项（即简单收益率），那么在程序中就可省略简单收益率的计算过程。程序中利用 NumPy 的 diff()函数计算股票差价是因为 diff()函数可以用数组后一个位置上的数据减去当前位置的数据之差作为结果数组中该位置上的数据。此外，若要使用 log()函数计算每个交易日收盘价的对数收益率，则应该检查变量 df 中的收盘价格数据列，以确保该数据列不含有零和负值（正常情况下的股票收盘价不会有零和负值，但在填充股票停牌期间的收盘价格数据时可能会出现零或负值）。

11.2.2　单只股票和市场平均收益率比较

股票市场的平均收益率通常是指股票价格指数（不同指数所涵盖的股票对象有差别）的收益率。我国目前使用的 A 股指数有几十种之多，其中包括上证指数、深圳成份指数、沪深 300 指数、中证 100 指数、上证 180 指数等。如果要比较单只股票收益率与市场平均收益率，那么可以根据研究的目的选择合适的市场指数作为比较对象。本节选择上海证券交易所交易股票“中国平安（股票代码 601318）”与上证指数比较彼此的收益率，具体比较“中国平安”单只股票与上证指数在近 3 年的月收益率和累计收益率。

第 1 步：根据计算月收益率的需要，选择下载股票价格的月线数据：

```
In [1]: import tushare as ts
In [2]: df1 = ts.get_k_data('601318',start='2016-01-01',end = '2019-01-1',ktype = 'M')
```

```
In [3]: df1.head()
Out[3]:
         date    open   close    high     low      volume    code
0  2016-01-29  33.574  28.272  33.574  26.951  13349153.0  601318
1  2016-02-29  28.253  27.176  29.649  26.052   6969416.0  601318
2  2016-03-31  27.073  29.799  31.663  26.755  14540561.0  601318
3  2016-04-29  29.818  30.015  30.558  29.078   8842452.0  601318
4  2016-05-31  29.987  30.652  31.382  28.994   6917618.0  601318
```

这是股票 601318 的月线行情数据形式，其中 open 是月初第一个交易日的开盘价，close 是月末最后一个交易日的收盘价，high 是整个月的交易最高价格，low 是整个月的交易最低价格。

类似地，利用 get_k_data()接口函数下载上证指数的月线数据：

```
In [4]: df2 = ts.get_k_data('sh', start = '2016-01-01', end = '2019-01-1', ktype = 'M')
In [5]: df2.head()
Out[5]:
         date     open    close     high      low        volume code
0  2016-01-29  3536.59  2737.60  3538.69  2638.30  4.007772e+09   sh
1  2016-02-29  2730.98  2687.98  2933.96  2638.96  3.013291e+09   sh
2  2016-03-31  2688.38  3003.92  3028.32  2668.76  5.084600e+09   sh
3  2016-04-29  2997.09  2938.32  3097.16  2905.05  3.786963e+09   sh
4  2016-05-31  2940.39  2916.62  3004.42  2780.76  2.853848e+09   sh
```

第 2 步：合并上证指数和中国平安股票的数据。因为单只股票存在停牌的情况，因此应该以上证指数的交易日为基准来合并两个数据序列：

```
In [6]: df = df2[['date', 'close']] .copy()
In [7]: df.rename(columns = {'close':'indclose'},inplace = True)
In [8]: df.head()
Out[8]:
        date  indclose
0  2016-01-29  2737.60
1  2016-02-29  2687.98
2  2016-03-31  3003.92
3  2016-04-29  2938.32
4  2016-05-31  2916.62
```

根据上证指数生成一个新的 DataFrame 对象 df，为避免在同一个 DataFrame 中出现列名称冲突，合并前将上证指数的 close 列名称更改为 indclose，再将 df1 的 close 列合并到 df 中：

```
In [9]: df = pd.merge(df[['date', 'indclose']], df1[['date', 'close']],
                      on = 'date', how = 'left')
```

为了消除单只股票停牌造成的缺失数据项，可以采用缺失数据项最近的前或后的有效数据填充各缺失数据项。因 get_k_data()函数的返回值是以时间先后排列各交易日数据的，所以如果非数据起始日因股票停牌而出现缺失值 NaN，那么比较合适的处理方法是用停牌前的最近有效数据填充 NaN 数据项。如果在数据序列首项出现缺失值，那么应采用距离缺失数据项最近的有效数据向前填充。因此在进行第 3 步操作前，需要查看数据序列中缺失值的位置并进行恰当的填充：

```
In [10]: df.fillna (method = 'ffill')
```

第 3 步：分别计算股票和指数的月收益率及其累计收益率。

```
In [11]: import numpy as np
In [12]: df['stk_log_ret'] = np.round(np.log(df['close']/df['close'].shift(1)), 4)
```

```
In [13]: df['ind_log_ret'] = np.round(np.log(df['indclose']/df['indclose'].shift(1)), 4)
In [14]: df['stk_log_ret_cum'] = df['stk_log_ret'].cumsum()
In [15]: df['ind_log_ret_cum'] = df['ind_log_ret'].cumsum()
```

第 4 步：绘制图形分别输出单只股票与指数的月收益率及其累计收益率。因为要比较两个指标的差异，图表须采用双纵轴点线图输出数据序列。考虑到图形横坐标数据量较多，坐标点 ticks 标签需要适当倾斜。在绘制图表输出时，最好对各序列数据点线图的颜色参数进行明确设置，以增强图形的可辨识性。

为了增强单只股票收益率与同期市场指数收益率（即平均收益率）的对照性，本程序采用双纵坐标格式输出的图表，如图 11.3 所示。图 11.3 中的横坐标是两个数据序列同期的交易日，纵坐标分别表示两者的收益率值的范围。程序中语句 ax2 = ax.twinx()的功能是创建一个独立的纵轴而共享横轴的双坐标图，其中 twinx()是 Matplotlib 的内建函数。

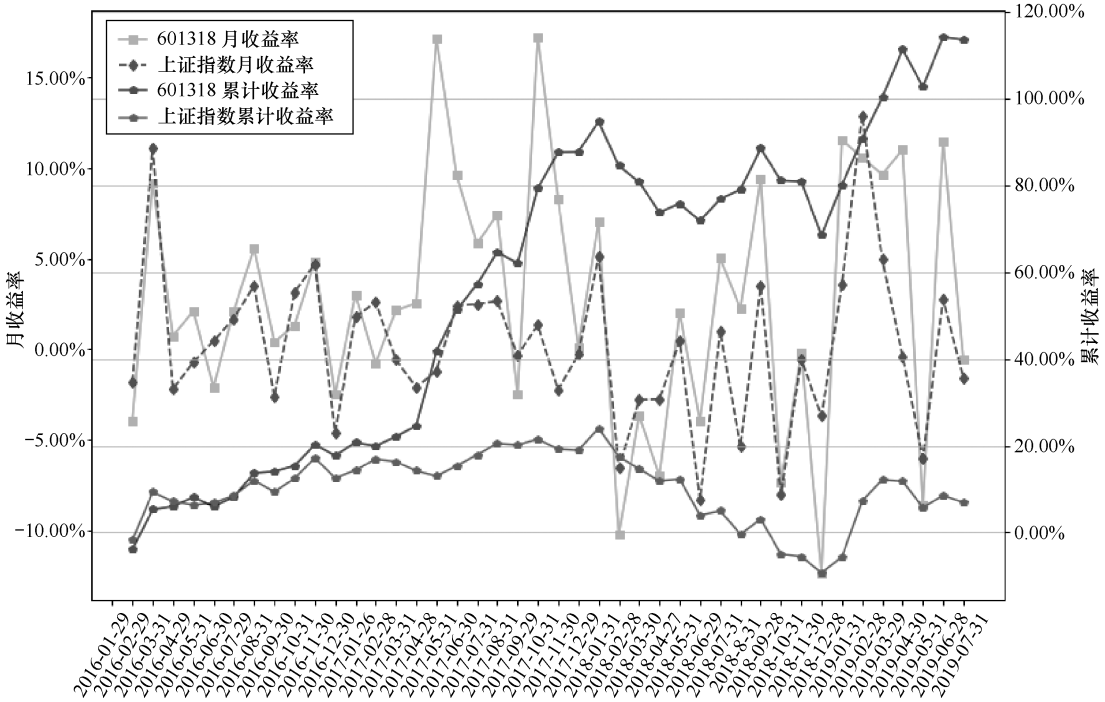


图 11.3　单只股票收益率与同期市场平均收益率比较

本例的完整程序如下：

```
import tushare as ts
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import ticker

startday = '2016-01-01'
endday = '2019-08-19'
tscode = '601318'
tsindx = 'sh'
df1 = ts.get_k_data(tscode, start = startday, end = endday, ktype = 'M')
df2 = ts.get_k_data(tsindx, start = startday, end = endday, ktype = 'M')
```

```
        df = df2[['date', 'close']].copy()
        df.rename(columns={'close': 'indclose'}, inplace = True)
        df = pd.merge(df[['date', 'indclose']],
                      df1[['date', 'close']], on = 'date', how = 'left')
        df.fillna(method = 'ffill')
        # 计算月收益率和累计收益率
        df['stk_log_ret'] = np.round(np.log(df['close']/df['close'].shift(1)), 4)
        df['ind_log_ret'] = np.round(np.log(df['indclose']/df['indclose'].shift(1)), 4)
        df['stk_log_ret_cum'] = df['stk_log_ret'].cumsum()
        df['ind_log_ret_cum'] = df['ind_log_ret'].cumsum()
        # 绘制累计收益率曲线
        ymajorFormatter = ticker.FormatStrFormatter('%.2f%%')  # 设置 y 轴标签文本的格式
        plt.rcParams['font.sans-serif'] = ['SimHei']  # 用来正常显示中文标签
        fig, ax = plt.subplots(figsize = (12, 9))
        fig.subplots_adjust(bottom = 0.2)
        plt.ylabel('月收益率')
        plt.xticks(rotation = 60)
        ax.yaxis.set_major_formatter(ymajorFormatter)  # 显示百分比
        ax.plot(df.date, df['stk_log_ret']*100, '-cs', lw = 1.5, label = tscode+' 月收益率')
        ax.plot(df.date, df['ind_log_ret']*100, '--md', lw = 1.5, label = '上证指数月收益率')
        plt.legend(loc = 'lower center')
        ax2 = ax.twinx()    # 这是双坐标的关键一步
        ax2.yaxis.set_major_formatter(ymajorFormatter)  # 显示百分比
        ax2.plot(df.date, df['stk_log_ret_cum']*100,
                 '-rp', lw=1.5, label=tscode+' 累计收益率')
        ax2.plot(df.date, df['ind_log_ret_cum']*100, '-gp', lw = 1.5,
                 label = '上证指数累计收益率')
        plt.legend(loc = 'upper right')
        plt.ylabel('累计收益率')
        plt.grid(True)
        plt.show()
```

11.2.3 历史波动率计算

股票的历史波动率是指股票投资回报率(即收益率)在过去一段时间内所表现出来的波动率，它由标的股票价格过去一段时间的历史时间序列数据反映，主要用于评估投资风险的大小。历史波动率的计算模型有传统波动率模型、高低价波动模型、R&S 模型、G&K 模型及 GARCH 模型等，其中传统波动率模型为历史波动率模型的基础模型，也是运用最广泛的波动模型之一。本节仅讨论传统波动率模型的 Python 实现方法。

传统波动率的计算主要包括如下三个步骤。

第 1 步：获得标的股票在指定交易期（如日、周、月等）的价格（股票价格的周线、月线可以在日线的基础上通过计算得到，但有些平台提供周线和月线数据）。

第 2 步：对于每个交易期计算该期末的股价与前一个交易期末的股价之比的自然对数（即计算对数收益率）。

第 3 步：计算这些对数值的标准差［NumPy 有内建标准差函数 `std()`］。标准差的计算公式为

$$\sigma=\sqrt{\frac{\sum_{i=1}^{n}\left(X_i-\bar{X}\right)^2}{n-1}}$$

式中，σ 是对数收益的标准差，X_i 是第 i 期的对数收益率，$\bar{X}$ 是计算波动率时间周期内对数收益率的平均值，n 是观察值的数量（即样本数量）。由于我国每年约有 250 个交易日，所以以时间间隔为每天（X_i 为日收益率）计算的标准差应乘以 $\sqrt{250}$，即在 n 期内股票价格的历史（年）波动率为 $\hat{\sigma}=\sigma\sqrt{250}$。

```
# 股票传统波动率计算
import tushare as ts
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import ticker
startday = '2018-01-01'
endday = '2019-08-19'
tscode = '601318'
df = ts.get_k_data(code=tscode, start=startday, end=endday)
# 计算对数收益率
df['log_price'] = np.round(np.log(df['close']), 4)
df['log_ret'] = df['log_price'].diff()  # 计算对数收益率，所有价格取对数后两两间的差值
df.dropna()  # 填充第一行无效数据
df['volatility'] = np.round(df['log_ret'].rolling(
    window=5, center=False).std()*np.sqrt(250), 4)  # 5 日移动平均波动率
df.fillna(0, inplace=True)
data = {'date': pd.Series(df['date'].values),
        'close': pd.Series(df['close'].values),
        'volatility': pd.Series(df['volatility'].values)
        }
df2 = pd.DataFrame(data)  # 重新编排索引，有利于设置坐标刻度
plt.rcParams['font.sans-serif'] = ['SimHei']  # 用来正常显示中文标签

def format_date(x, pos):
    if x < 0 or x > len(date_tickers)-1:
        return('')
    return(date_tickers[int(x)])

date_tickers = df2['date'].values  # 生成收盘价横轴的刻度名字
fig = plt.figure(figsize=(16, 9))
ax1 = fig.add_subplot(2, 1, 1)  # 第一个子图绘制波动率
ax1.plot(df2['volatility']*100)
ax1.set_xlim([0, len(date_tickers)+15])
ax1.xaxis.set_major_formatter(ticker.FuncFormatter(format_date))
ax1.xaxis.set_major_locator(ticker.LinearLocator(15))
plt.setp(plt.gca().get_xticklabels(), rotation = 30, horizontalalignment = 'right')
ax1.grid(True)
plt.ylabel('波动率 %')
```

```
        plt.title(tscode)
        ax2 = fig.add_subplot(2, 1, 2)  # 第二个子图绘制收盘价
        ax2.plot(df2['close'])
        ax2.set_xlim([0, len(date_tickers)+15])
        ax2.xaxis.set_major_formatter(ticker.FuncFormatter(format_date))
        ax2.xaxis.set_major_locator(ticker.LinearLocator(15))
        plt.setp(plt.gca().get_xticklabels(), rotation = 30, horizontalalignment = 'right')
        ax2.grid(True)
        plt.ylabel('交易价格')
        plt.show()
```

历史波动率的时间区间通常是以年为单位（故称年化波动率）。波动率本是一个观察时点的数据，选择不同的观察周期计算波动率的结果是不一样的。然而，选择合适的观察期 n（样本数量）并不容易（因为目前还没有严格的理论依据），但通常选择 5 日、15 日、30 日等作为最小观察期。本程序选择 5 日作为观察周期，采用 5 日移动平均值的计算方法求整个时间段内各时点的波动率。股票（601318）在过去一年多时间内的波动率计算结果如图 11.4 所示。

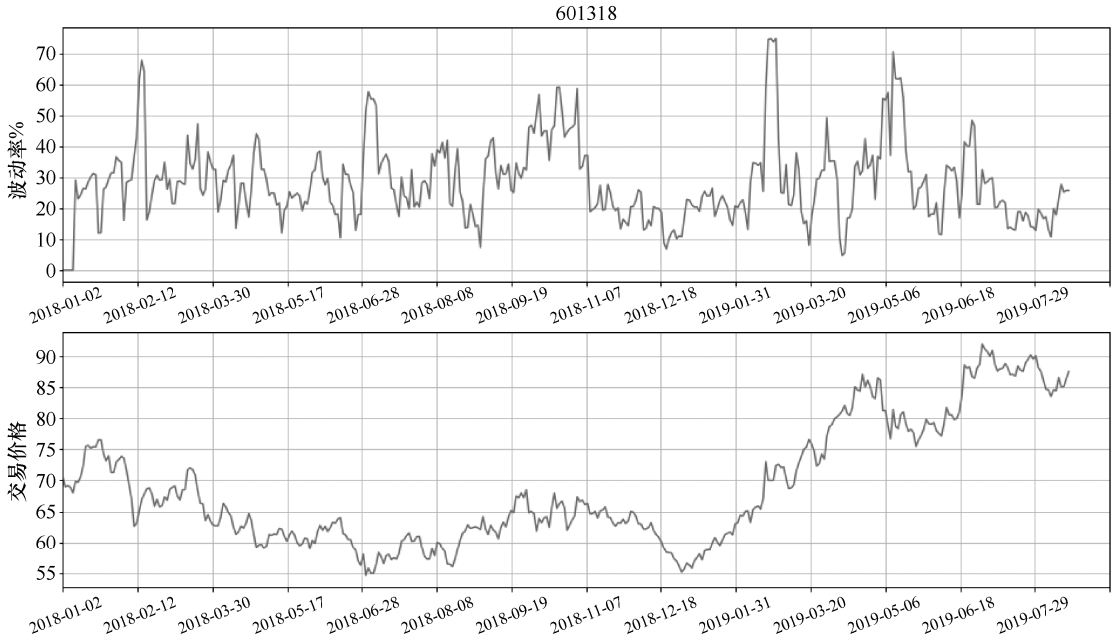


图 11.4　股票价格日波动率

11.2.4　股票收益率相关性分析

同一个市场中不同股票的涨跌幅度总会有或大或小的差异，即使是在牛市（指整个股票市场的股票平均价格发生较长时间和较大幅度的上涨）中也会有价格大幅度下跌的股票，在熊市（指整个股票市场的股票平均价格发生较长时间和较大幅度的下跌）中也会有价格大幅度上涨的股票。不同股票的价格涨跌差异性自然会增加股票投资者获取收益的难度。实践经验表明，股票投资组合中的个股之间的相关性（指股票收益率的相关性）越低，该投资组合抵抗市场风险的能力越强。

相关系数是用来评价两个数据序列相关性的关键指标，其取值区间是[−1, 1]。相关系数等于

0 时，表示两个数据序列不相关；相关系数小于 0 时，表示两个序列存在负相关性；相关系数大于 0 时，表明两个序列存在正相关性。相关系数的绝对值越大（即越接近于 1），两个序列的相关性越强。特别地，当相关系数的绝对值等于 1 时，称这两个序列是线性相关的。

股票价格指数主要用来表明若干只股票价格的加权和数。在我国特别受到股票市场参与者关注的指数包括上证指数、深证成指和创业板指三个指数，这三个指数堪称我国股票市场的“温度计”，稍有经验的股票交易者都会根据这三个指数的变化趋势来寻找股票交易机会，以便获取比较理想的收益。因此，本节计算的股票相关系数也考察了这三个指数之间的相关性，并且计算了这些指数与所关注个股间的相关系数。

为增强相关系数的可信度，在计算不同股票收益率相关系数时使用的行情数据需要体现分红配股产生的影响。目前获取 Tushare 平台提供的股票复权数据还比较困难，但获取 Yahoo 金融平台提供的股票复权数据比较容易。为了获取 Yahoo 金融平台的股票数据，本机须安装 pandas-datareader 模块（在 Anaconda 的命令行执行命令 pip install pandas-datareader 后，在 Spyder 环境下就可用命令 import pandas_datareader.data 导入需要的提取数据子模块），利用该模块的 DataReader()函数才能获取 Yahoo 金融平台提供的股票行情数据。DataReader()函数的参数说明如表 11.2 所示。

表 11.2　DataReader()函数的参数

参　数	格　式	说　明
name	字符串	股票代码
data_source	字符串	数据源
start	字符串/日期时间/None	起始时间
end	字符串/日期时间/None	截止时间

利用 DataReader()函数获取 Yahoo 金融平台的股票数据须导入 pandas_datareader.data 包：

```
In [1]: import pandas_datareader.data as web
```

例如，从 Yahoo 金融平台读取阿里巴巴公司的股票数据的操作命令如下：

```
In [2]: alibaba = web.DataReader("BABA", "yahoo", '2018-1-1', '2019-2-1')
In [3]: alibaba.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 273 entries, 2018-01-02 to 2019-02-01
Data columns (total 6 columns):
High         273 non-null float64
Low          273 non-null float64
Open         273 non-null float64
Close        273 non-null float64
Volume       273 non-null int64
Adj Close    273 non-null float64
dtypes: float64(5), int64(1)
memory usage: 14.9 KB
```

对比 Tushare 提供的股票行情信息，Yahoo 金融的数据格式与之有些差别，其中 Adj Close 列是经过复权调整的收盘价格，它是前复权收盘价格。若要提取国内市场的股票数据，则函数的第一个参数为股票编号（沪市股票代码后加上.ss，深市股票代码后加上.sz），例如：

```
In [4]: df1 = web.DataReader("601318.ss", "yahoo", '2018-1-1', '2019-2-28')
```

```
In [5]: df2 = web.DataReader("000988.sz", "yahoo", '2018-1-1', '2019-2-28')
```

其中 df1 是上海证券交易所股票代码为 601318 的行情数据，df2 是深圳证券交易所股票代码为 000988 的行情数据。

如下程序可以根据用户需要计算任意给定的多只股票（用列表列出股票代码）之间的相关系数，并且可以将累计收益率绘制成二维图形以便查看收益率的变化情况。为了体现 2015 年上半年出现的牛市行情和最近 3 年多的熊市行情数据相关性，程序中将 2014 年 1 月 1 日设置为起始日期。此外，考虑到股票停牌期间股票价格保持不变，程序中采用停牌前最后交易日的收盘价填充停牌交易日的收盘价来计算对数收益率。

```
import tushare as ts
import pandas as pd
import numpy as np
import datetime
import pandas_datareader.data as web
import matplotlib.pyplot as plt

# 获得指数数据，确定交易日日期和点位数据
start_day = datetime.datetime(2014, 1, 1)  # 考察 2015 年的牛市行情和近 3 年多的熊市行情
end_day = datetime.datetime.today()
#  分别获取上证指数、深证成指和创业板指
dats = ts.get_k_data('sh', start_day.strftime('%Y-%m-%d'),
                end_day.strftime('%Y-%m-%d')).set_index('date').loc[:, ['close']]
datz = ts.get_k_data('sz', start_day.strftime('%Y-%m-%d'),
                end_day.strftime('%Y-%m-%d')).set_index('date').loc[:, ['close']]
datc = ts.get_k_data('cyb', start_day.strftime('%Y-%m-%d'),
                end_day.strftime('%Y-%m-%d')).set_index('date').loc[:, ['close']]
dats['szindx'] = datz['close']
dats['cybindx'] = datc['close']
dats.rename(columns = {'close': 'shindx'}, inplace = True)
# 创建股票池分析它们的相关性
st_list = ['600519.ss', '601318.ss', '601939.ss',
           '000001.sz', '000513.sz', '300003.sz']
for st in st_list:  # 挨个获取每只股票的数据，并存储调整后的收盘价
    data = web.DataReader(st, 'yahoo', start_day,
                          end_day).loc[:, ['Adj Close']]
    data.rename(columns = {'Adj Close': st.split('.')[0]}, inplace = True)
    dats = pd.merge(dats, data, how = 'outer', left_index = True,
                    right_index = True)  # 注意 Adj 和 Close 之间有一空格
# 用停牌前最后交易日的收盘价填充停牌交易日的收盘价
dats.fillna(method = 'ffill', inplace = True)
# 计算每日对数收益率
dats['sh_ret'] = np.round(np.log(dats['shindx']/dats['shindx'].shift(1)), 4)
dats['sz_ret'] = np.round(np.log(dats['szindx']/dats['szindx'].shift(1)), 4)
dats['cyb_ret'] = np.round(np.log(dats['cybindx']/dats['cybindx'].shift(1)), 4)
for st in st_list:
    dats[st.split('.')[0]+'ret'] = np.round(np.log(dats[st.split('.')
        [0]]/dats[st.split('.')[0]].shift(1)), 4)
```

```
dats.fillna(0, inplace=True)
col_name = dats.columns.values.tolist()   # 第一次出现的位置
ret_col = []
for x in col_name:
    if ('ret' in x):
        ret_col.append(x)
ret_dats = dats[ret_col]

# 累计收益曲线绘制函数
def cumulative_returns_plot(name_list):
    if len(name_list) == 0:
        print('没有绘图数据')
        return
    else:
        CumReturns = pd.DataFrame()
        for name in name_list:
            CumReturns[name] = np.round(ret_dats[name].cumsum()*100, 2)
    CumReturns.plot()
    plt.legend()
    plt.xticks(rotation = 30)
    plt.show()

correlation_matrix = np.round(ret_dats.corr(), 4)  # 计算相关系数矩阵
correlation_matrix.to_excel('xiangguanjuzhen.xlsx')
# 输出相关矩阵，绘制累计收益率曲线
print(correlation_matrix)
cumulative_returns_plot(['sh_ret','sz_ret','cyb_ret','600519ret','601318ret','300003ret'])
```

相关系数矩阵数据如表 11.3 所示。

表 11.3　相关系数矩阵数据表

	sh_ret	sz_ret	cyb_ret	600519ret	601318ret	601939ret	000001ret	000513ret	300003ret
sh_ret	1.0	0.9216	0.7372	0.5162	0.7052	0.6397	0.0866	0.5034	0.5079
sz_ret	0.9216	1.0	0.8679	0.4921	0.5765	0.4356	0.0914	0.5663	0.5738
cyb_ret	0.7372	0.8679	1.0	0.3277	0.3449	0.1927	0.0722	0.5365	0.6315
600519ret	0.5162	0.4921	0.3277	1.0	0.5058	0.367	0.1076	0.3633	0.2837
601318ret	0.7052	0.5765	0.3449	0.5058	1.0	0.6672	0.1131	0.3244	0.2698
601939ret	0.6397	0.4356	0.1927	0.367	0.6672	1.0	0.0667	0.2037	0.1857
000001ret	0.0866	0.0914	0.0722	0.1076	0.1131	0.0667	1.0	0.0479	0.0515
000513ret	0.5034	0.5663	0.5365	0.3633	0.3244	0.2037	0.0479	1.0	0.4873
300003ret	0.5079	0.5738	0.6315	0.2837	0.2698	0.1857	0.0515	0.4873	1.0

从表 11.3 可以看出，上证指数和深证成指之间的相关性较强，创业板指与深证成指之间的相关性要高于创业板指与上证指数之间的相关性。上海证券交易所交易的股票（股票编号以 60 开头）与上证指数之间的相关性要比它与深证成指、创业板指之间的相关性稍强，而深圳交易所交易的股票（股票编号以 00 或 30 开头）与深证成指之间的相关性也要比它与上证指数之间的相

关性稍强。但是，并非同在一个交易所交易的股票都有较强的相关性。例如，股票 600519 与股票 601393 之间的相关性就较弱。另外，单只股票与其所属市场的指数之间的相关性也可以比较弱，如股票 000001 与深证成指之间的相关性就非常弱。股票之间的相关性更具多样性。例如，股票 300003 在过去 5 年与实例中其他所有股票的相关性都非常低，实际上股票 300003 除 2014 年 1 月到 2016 年 5 月中旬间的价格走势比较同步上述三大指数的走势外，自 2017 年 1 月开始的走势非常具有独立性（俗称独立行情），股票 600519 也似乎与其他股票之间的相关性都非常低，其价格走势特征也可以从如图 11.5 所示的股票价格累计收益率走势得到印证。

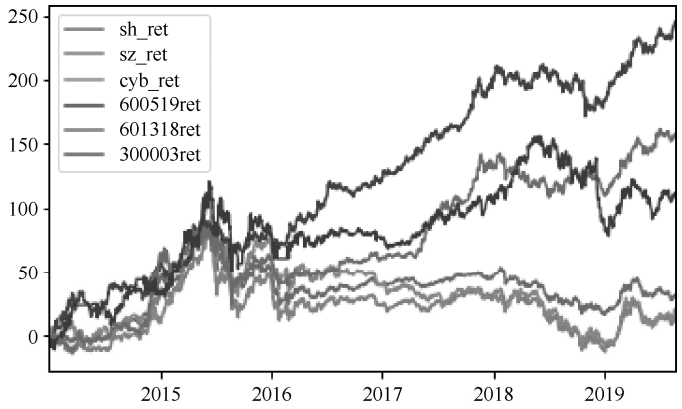


图 11.5　股票价格累计收益率

11.3　股票技术指标计算

股票技术指标是相对于公司基本面指标而言的股票价格走势评估指标。基本面分析着重于对上市公司的财务质量和成长能力等一般经营状况、行业发展趋势等因素进行（定性与定量）分析，并以此来判断公司股票的投资价值。技术分析则通过股票价格图表或与股票价格变化相关联的技术指标数据来评估股票价格的未来走势，通过量化技术指标来推测股票价格的变动趋势。目前比较通用的技术指标大都是由股票历史价格序列和成交量序列等数据计算而来的。

11.3.1　价格趋势分析

股票价格移动平均线是一种反映股票价格走势的曲线段。投资者通过观察移动平均线的走势来判断股票价格是上涨还是下跌，从而决定买入股票或卖出股票。当股票价格变动具有明显趋势时，移动平均线的趋势指引功能相当显著。相对于简单移动平均线而言，异同移动平均线（MACD）对股票价格趋势变动的反映更为有效，因为它吸收了简单移动平均线的趋势性优点，克服了简单移动平均线假信号频繁的缺陷。

在实际应用中，MACD 首先需要计算出一条快速移动平均线（即 EMA1）和一条慢速移动平均线（即 EMA2），再计算这两个数值（快、慢速线）之间的离差值（DIF），最后计算 DIF 的 N 周期平滑移动平均线 DEA（也称 MACD、DEM）线。

EMA（Exponential Moving Average）是指数移动平均值，它是以指数式递减加权的移动平均，其一般表达式为

$$\mathrm{EMA}_{\mathrm{today}} = \alpha\,\mathrm{Price}_{\mathrm{today}} + (1-\alpha)\,\mathrm{EMA}_{\mathrm{yesterday}}$$

式中，α 是平滑指数，一般取值为 $2/(N+1)$。

在计算 MACD 指标时，EMA 计算中的 N 一般选取 12 和 26(天)，因此 α 相应为 2/13 和 2/27。在 EMA 指标计算公式中，每天价格的权重系数以指数等比形式缩小，时间越靠近当今时刻的价格，对应的权重越大，这说明 EMA 函数对近期的价格加强了权重比，以便及时地反映近期价格的波动情况。

以 EMA1 的参数为 12（日）、EMA2 的参数为 26（日）、DIF 的参数为 9（日）为例，MACD 指标的计算过程如下。

① 计算移动平均值（EMA）：

EMA(12) = 今日收盘价×2/13 + 昨日 EMA(12)×11/13

EMA(26) = 今日收盘价×2/27 + 昨日 EMA(26)×25/27

② 计算离差值（DIF）：

DIF = 今日 EMA(12) − 今日 EMA(26)

③ 计算 DIF 的 9 日 EMA：

今日 DEA(MACD) = 今日 DIF×2/10 + 昨日 DEA×8/10

根据离差值计算其 9 日的 EMA（即离差平均值）便是股票价格走势的 MACD 值。

如下程序通过 pandas_datareader.data 模块的 DataReader()函数获取 Yahoo 金融平台提供的股票日线行情，利用行情 DataFrame 的[Adj Close]序列来计算 macd 指标。程序的输出结果如图 11.6 所示。

```
import numpy as np
import datetime
import matplotlib.pyplot as plt
import pandas_datareader.data as web
from matplotlib.gridspec import GridSpec
start = datetime.datetime(2018, 6, 1)
end = datetime.datetime.today()
stock_name = '601318.ss'
df = web.DataReader(stock_name, 'yahoo', start, end)

def df_EMA(prices, N):
    ema = []
    k = len(prices)
    if k > 0:
        for i in range(k):
            if i == 0:
                ema.append(prices[i])
            else:
                ema.append((2*prices[i]+(N-1)*ema[i-1])/(N+1))
    return(ema)

def df_MACD(df, short = 12, long = 26, M = 9):
    fast = df_EMA(df['Adj Close'].values, short)
    slow = df_EMA(df['Adj Close'].values, long)
    if len(fast) > 0:  # & len(slow)>0:
        df['Fast'] = np.round(np.array(fast), 2)
```

```
            df['Slow'] = np.round(np.array(slow), 2)
            df['DIF'] = df['Fast']-df['Slow']
            df['DEA'] = np.round(np.array(df_EMA(df['DIF'].values, M)), 2)
            df['MACD'] = 2*(df['DIF']-df['DEA'])
            return(df)
        else:
            print('no data,no MACD')

df_MACD(df, 12, 26, 9)
figure = plt.figure(figsize = (9, 6))
plt.rcParams['font.sans-serif'] = ['SimHei']  # 用来正常显示中文标签
plt.rcParams['axes.unicode_minus'] = False    # 正常显示负号
gs = GridSpec(3, 1)
ax1 = plt.subplot(gs[:2, :])
ax2 = plt.subplot(gs[2, :])
ax1.plot(df.index.values, df['Fast'])
ax1.plot(df.index.values, df['Slow'])
ax1.set_title(stock_name.split('.')[0], fontsize = 20)
df['up'] = (df['MACD'] >= 0)
ax2.bar(df.query('up == True').index.values,
        df.query('up == True')['MACD'], color = 'r',
        width=0.75, alpha=0.85)
ax2.bar(df.query('up == False').index.values,
        df.query('up == False')['MACD'], color = 'g',
        width=0.75, alpha=0.85)
plt.show()
```

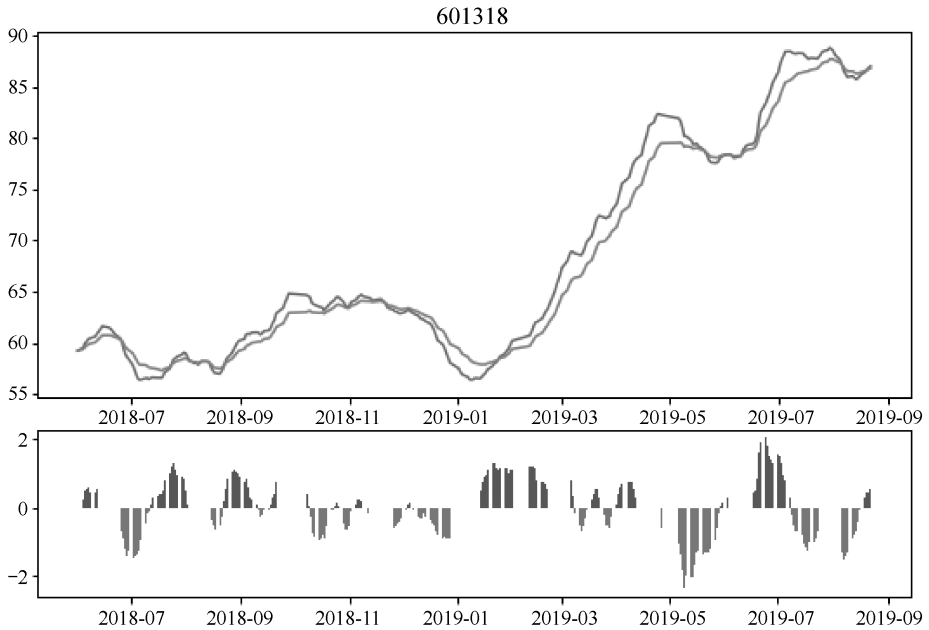


图 11.6　股票 MACD 指标

从理论上说，如果股票价格在持续的涨势中 12 日 EMA 线在 26 日 EMA 线之上，那么其间的正离差值（+DIF）会越来越大（对应的 MACD 红色柱子越长）；反之，在跌势中离差值可能变为负数（-DIF），并且会负得越来越大（对应的 MACD 绿色柱子越长），而在行情开始转向时正负离差值将会缩小。指标 MACD 正是利用正负的离差值（±DIF）与离差值的 N 日平均线（N 日 EMA）的交叉来表达行情变化节点的，即以快、慢速移动线的交叉原理来界定买入和卖出股票的信号。从实践来看，快线从下往上穿越慢线时可以视为买入信号，快线从上往下穿越慢线时可以视为卖出信号，但股票价格的涨跌在大多数期间是在某个价格中枢上下振荡的，快、慢线交叉频率会比较高，且不同周期（日、周、月等）下的 MACD 指标会出现较大的不一致性，因此其买卖信号的指导意义会降低。

11.3.2　超卖超买分析

在股市中人们习惯称对某只股票的过度买入为超买，称对某只股票的过度卖出为超卖。股市经常会出现因某种消息的传播而使投资者对股票市场做出强烈的反应，以致引起整个股市或个股价格出现过分的上升或下跌而产生超买超卖现象。当投资者的情绪平静下来后，超买超卖所造成的影响会逐渐得到适当的调整。因此，超买之后股价就会出现一段时间的回落，超卖之后股价则会出现一波相当程度的回升。

随机指数（KDJ）是投资者经常用来测算超买超卖现象的技术指标。指标 KDJ 的计算包括以下两个步骤。

① 计算周期（n 日、n 周等）的未成熟随机指标值 RSV：

$$\mathrm{RSV} = (C_n - L_n)/(H_n - L_n) \times 100$$

式中，C_n 为第 n 日的收盘价；L_n 为 n 日内的最低价；H_n 为 n 日内的最高价。RSV 值始终在 1～100 间波动。

② 计算 K 值、D 值、J 值：

$$\text{当日 } K \text{ 值} = 2/3 \times \text{前一日 } K \text{ 值} + 1/3 \times \text{当日 RSV}$$

$$\text{当日 } D \text{ 值} = 2/3 \times \text{前一日 } D \text{ 值} + 1/3 \times \text{当日 } K \text{ 值}$$

$$\text{当日 } J \text{ 值} = 3 \times \text{当日 } D \text{ 值} - 2 \times \text{当日 } K \text{ 值}$$

若无前一日的 K 值与 D 值，则可分别用 50 来代替。式中的平滑因子 1/3 和 2/3 是可以人为选定的，不过目前已经约定俗成地固定为 1/3 和 2/3。根据选用计算周期的不同，KDJ 指标包括日 KDJ 指标、周 KDJ 指标、月 KDJ 指标、年 KDJ 指标和分钟 KDJ 指标等类型，其中日 KDJ 指标和周 KDJ 指标的指引意义相对较强。

KDJ 指标的指引规则如下：当指标线 D 小于 20 时表示进入超卖区间，它被视为买入信号。当指标线 D 大于 80 时进入超买区间，它被视为卖出信号。当指标线 K 自下向上突破指标线 D 时形成 KDJ 指标金叉（买入信号），当指标线 K 自上向下穿破指标线 D 时形成 KDJ 指标死叉（卖出信号）。

如下程序实现了周期 $n = 9$ 的 KDJ 指标的计算，程序执行的输出结果如图 11.7 所示。

```
import pandas as pd
import datetime
from matplotlib.dates import DateFormatter
import matplotlib.pyplot as plt
import pandas_datareader.data as web
```

```
start = datetime.datetime(2019, 3, 1)
end = datetime.datetime.today()
stock_name = '600519.ss'
df = web.DataReader(stock_name, 'yahoo', start, end)
df['minlow'] = df['Low'].rolling(window = 9).min()
df['maxhigh'] = df['High'].rolling(window = 9).max()
df['rsv'] = (df['Close']-df['minlow'])/(df['maxhigh']-df['minlow'])*100
df['k'] = df['rsv'].ewm(span = 2, ignore_na = True, adjust = True).mean()
df['d'] = df['k'].ewm(span = 2, ignore_na = True, adjust = True).mean()
df['j'] = 3*df['k']-2*df['d']

def format_date(k, pos):  # 设置坐标格式
    if k < 0 or k > len(xs)-1:
        return ''
    return xs[int(k)]

xs = pd.to_datetime(df.index)
figure, ax = plt.subplots(figsize = (7, 4))
ax.xaxis.set_major_formatter(DateFormatter('%b %d, %Y'))
df.loc[:, ['k', 'd', 'j']].plot(ax = ax, lw = 1.5, grid = True)
ax.set_title(stock_name.split('.')[0], fontsize = 20)
plt.xticks(rotation = 30)
ax.legend()
plt.show()
```

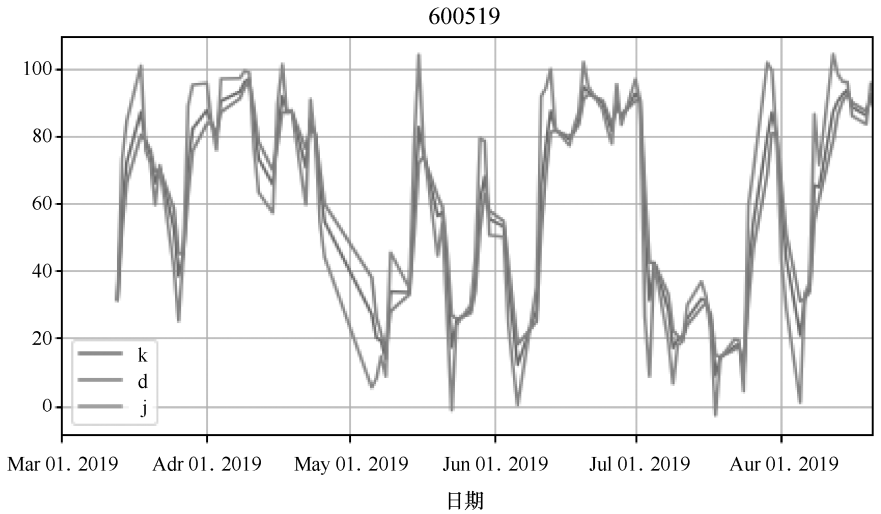


图 11.7　股票 KDJ 指标

以上两个关于股票技术分析指标（MACD 和 KDJ）的计算只是为了介绍 Python 语言的一种应用，而非给读者做任何股票交易技术指标的推荐。虽然股票市场的程序化交易研究与实践方兴未艾，但开发设计一个成功率较高的交易系统是一项非常复杂的工作，并且随着市场运行的推移而需要不断更新指标及其参数的设置。事实上，构建一个成功率高于失败率的交易系统在很大程度上是一项艺术工作，远非凭借一套技术指标就能为交易者构建带来稳定收益的交易系统。

11.4　宏观经济数据分析

宏观经济是指整个国民经济或国民经济总体及其经济活动和运行状态，包括一个国家的国民经济总量、国民经济构成、产业发展阶段与产业结构、经济发展程度等内容。宏观经济指标是体现宏观经济情况的一种方式，主要指标包括国内生产总值、通货膨胀与紧缩、投资、消费、金融、财政及汇率等。本节以我国近 25 年的国内生产总值、国内生产总值（年）增长率、人均国民生产总值、汇率等指标为例，分别介绍利用 Python 程序对宏观经济数据进行简单分析的方法。

11.4.1　数据准备

世界银行公开数据平台（https://data.worldbank.org.cn/）目前提供全球 264 个国家的宏观经济数据，有些经济指标包括过去 50 年的数据，但早于 1990 年的数据大多不够完整，有些指标甚至仅包括自 2000 年至今的数据。

根据世界银行公开数据平台的指引（从平台下载财经数据的详细操作流程请参阅本书实验 9 中的相关说明），在平台网页上需要从 Country、Series 和 Time 三个数据集中选择需要关于哪个国家、哪项指标及该指标数据的时间区段以确定数据序列的选择范围。确定全部选项后才可以下载查询到的指标数据序列。从平台下载的数据可以保存为 Excel、CSV 和 TXT 三种格式的文件。这些指标数据包含的字段（列）通常较多，字段（列）基本形式如下：

```
Index(['Series Name', 'Series Code', 'Country Name', 'Country Code',
       '1969 [YR1969]', '1970 [YR1970]', …])
```

这样的列结构数据并不便于 Python 分析程序使用，还需要对数据进行清洗。例如，假设将查询的几个经济指标数据序列保存为 CSV 文件 gdpchain.csv，那么可以参照如下三个操作步骤生成一个便于 Python 分析程序使用的 DataFrame 对象数据。

第 1 步：定义一个函数（要保存为.py 文件）以便简化原数据序列的字段（列）名称。

```
df = pandas.read_csv('gdpchain.csv')  # gdpchain.csv 存在于当前工作目录下
def modifycolumns(df):
    cols = list(df.columns)
    for i in range(4,len(cols)):
        df.rename(columns = {cols[i]:cols[i].split(' ')[0]}, inplace = True)
    return(df)
df = modifycolumns(df)
```

这样得到的 df 中从 4 号列开始的列名称是原 df 对应列名称的全部年号，如'1969 [YR1969]'中的'1969'。

第 2 步：删除 df 中多余的空行和多余的列。第 1 步得到的 df 往往会存在多个空白行，这些空白行没有任何数据。例如，若只关心时间区段内某项指标的数据，则可为对应的'Series Name'重新设定一个更为简单的名字，同时提取并保存其对应各时期的数据。例如：

```
yearsgdp = df.columns[4:len(df.columns)]
df2 = pandas.DataFrame(yearsgdp,columns = ['year'])
df2['gdp'] = tuple(df.iloc[0,4:len(df.columns)])
```

这样构建的 DataFrame 对象 df2 中包含'year'和'gdp'两列（假设 df 中第一行是 gdp 指标数据），其中'year'列保存 df 的全部时间年号，'gdp'保存各年度的 gdp 值。可以照此方法处理其他各项经济指标的数据序列。

第 3 步：将清洗得到的数据保存为 CSV 格式的文件。设置 to_csv()中的参数 index=False 是避免将索引作为数据保存到 CSV 文件中，以便在下次读取该 CSV 文件数据时不会有多余的列值：

```
df2.to_csv('gdpchinaseries.csv',index = False)
```

通过上述三个操作步骤，一般可以从下载的原始数据序列中提取自己需要的数据。在 Python 程序需要处理这些数据时，只需利用函数 read_csv()读取文件数据就可以得到一个结构简洁的 DataFrame 对象：

```
mydf = pandas.read_csv('gdpchinaseries.csv')
```

本节使用的数据均是从 https://data.worldbank.org.cn/?locations=CN-XT 网页下载的，且后续实例使用的数据都是经过清洗的数据。

11.4.2 国内生产总值增长态势

为了探究我国近 25 年来的国内生产总值（GDP）总量及其增长率等变化情况，在世界银行公开数据平台选择“GDP（现价美元）”“GDP 增长率（年百分比）”“人均 GDP”“GDP 平减指数”“实际有效汇率”及“名义有效汇率”等指标。

（1）GDP 增长速度

由于相同产品的价格在不同年份往往会有不同，所以即使相同产量的商品在不同年份所表现的产值也会有差别，进而导致 GDP 数值有差别。以当年国内产品总量乘以当年产品价格得到的 GDP 被称为名义 GDP，当年国内产品总量乘以某参照年度产品的价格计算的 GDP 被称为实际 GDP。实际 GDP 剔除了名义 GDP 中因货币升值或贬值而导致的 GDP 增减成分。

为了使得不同年份的 GDP 具有可比性，通常选择某一年（称这个选定的年份为基年）的价格水平为基准来计算各年份的 GDP。基年的价格水平就是所谓的不变价格，按基年的不变价格计算出来的各年度最终产品的市场价值就是实际 GDP。名义 GDP 和实际 GDP 的关系为

$$\text{GDP 平减指数} = \text{名义 GDP} / \text{实际 GDP}$$

其中 GDP 平减指数也称 GDP 隐含缩减指数（implicit price deflator index for GDP），它是在给定的一年中名义 GDP 与该年实际 GDP 的比率。世界银行公开数据平台提供的 GDP 平减指数序列是选择 2015 年作为基年进行计算的结果，而平台提供的我国 GDP 数据序列是以现价美元计价的数值。

本节的实例是考察我国最近 25 年的 GDP（以现价美元计价）增长状况，以点线图的方式描述各年度的 GDP 变化（见图 11.8）。图 11.8 中左侧纵坐标表示 GDP 数值（以现价美元计价），右侧纵坐标表示 GDP 增长速度。从图 11.8 中不难发现，名义 GDP 与实际 GDP 的差别是以基年 2015 为分界点，在 2015 年前的各年度名义 GDP 比实际 GDP 明显要少，而 2015 年后的各年度名义 GDP 比实际 GDP 明显要多。这主要是因为此 GDP 序列数据是以现价美元计价的实际 GDP，而 2015 年（基年）后的美元贬值比较大。

计算 GDP 增长率及其变化情况的完整程序如下：

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import FormatStrFormatter

df = pd.read_csv('gdpchina19942017.csv')
# 计算名义 GDP 及其增长率：名义 GDP=实际 GDP*GDP 缩减指数
df['normgdp'] = np.round(df['current$']*np.round(df['defl']/100, 2), 4)
df['normgdprate'] = np.round((df['normgdp'] -
```

```
                                df['normgdp'].shift(1))/df['normgdp'].shift(1), 6)*100
    ymajorFormatter = FormatStrFormatter('%.2f%%')  # 设置 y 轴标签文本的格式
    fig, ax = plt.subplots(figsize = (12, 8))
    plt.rcParams['font.sans-serif'] = ['SimHei']  # 用来正常显示中文标签
    plt.xticks(df.index, df.year)
    plt.xticks(rotation = 45)
    ax.plot(df.index, df['current$'], '-rs', lw = 1.5, label = '实际 GDP(现价美元)')
    ax.plot(df.index, df['normgdp'], '--gd', lw = 1.5, label = '名义 GDP(现价美元)')
    plt.legend(loc = 'lower center')
    ax2 = ax.twinx()
    ax2.yaxis.set_major_formatter(ymajorFormatter)  # 显示百分比
    ax2.plot(df.index, df['rate'], '-yD', lw = 1.5, label = '实际 GDP 增长率')
    ax2.plot(df.index, df['normgdprate'], '-b*', lw = 1.5, label = '名义 GDP 增长率')
    plt.legend(loc = 'upper center')
    plt.title('中国 GDP 增长速度')
    plt.grid(True)
    plt.show()
```

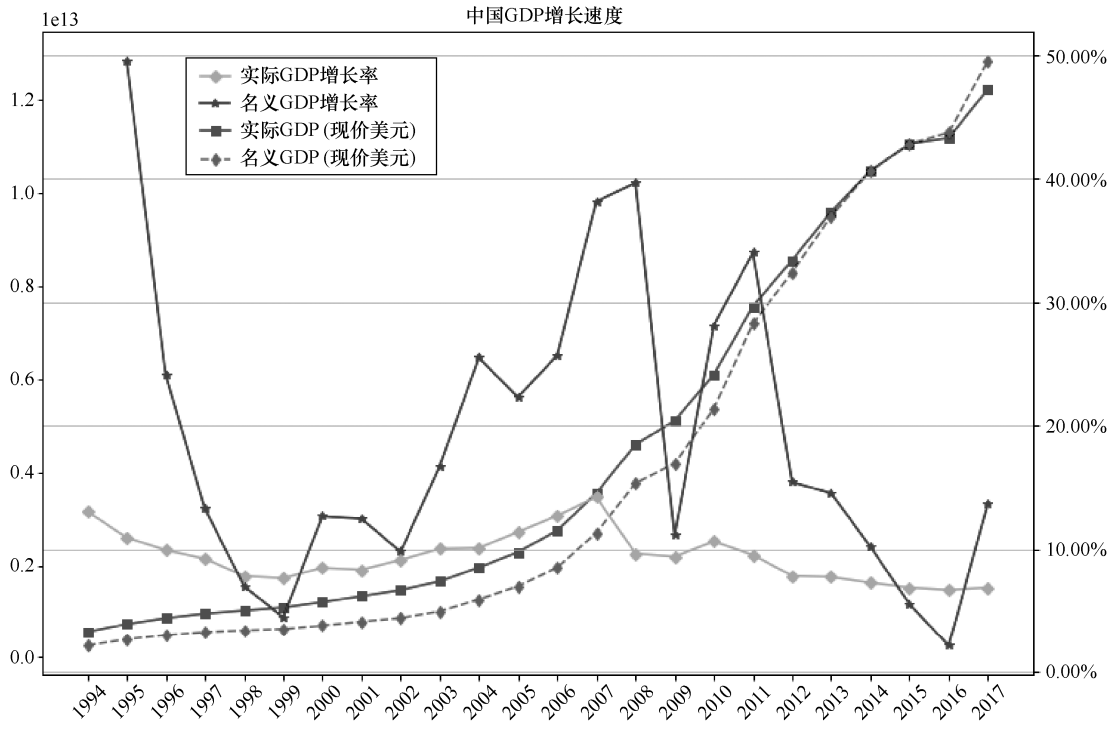


图 11.8　中国 GDP 增长速度

（2）人均 GDP 增长速度

实际人均国内生产总值（real GDP per capita，即实际人均 GDP）也是评价一个国家或地区的宏观经济运行状况的重要宏观经济指标之一。一个国家核算期内（通常是一年）实现的国内生产总值与这个国家的常住人口（或户籍人口）的比值即是人均国内生产总值。

从世界银行公开数据平台下载的中国人均 GDP 是以现价人民币计价（实际人均 GDP）的数值，它也是用现价美元计价的 GDP 计算得来的。为了分析过去若干年内我国人均 GDP 的变化情

况，本节采用如下关系式表示名义人均 GDP 与实际人均 GDP 的关系：

名义人均 GDP = 实际人均 GDP×GDP 平减指数

为方便对照我国同期的 GDP 增长状况，这里同样采用 2015 年作为 GDP 平减指数的基年，在实际人均 GDP 的基础上计算出名义人均 GDP 年度增长速度、实际人均 GDP 及其增长速度。

图 11.9 是我国近 25 年的人均 GDP 增长速度变化的基本情况。从图 11.9 中可以发现，1994 年至 2017 年间，大多数年份的名义人均 GDP 增速显著高于实际人均 GDP 增速，这表示大多数年份的通货膨胀比较显著，自 2000 年至今，仅有 2002 年、2009 年和 2015 年三年的名义人均 GDP 增速与实际人均 GDP 增速比较接近。程序中使用的数据文件 gdpchina19942017.csv 是经过清洗后的名义人均 GDP 数据文件。

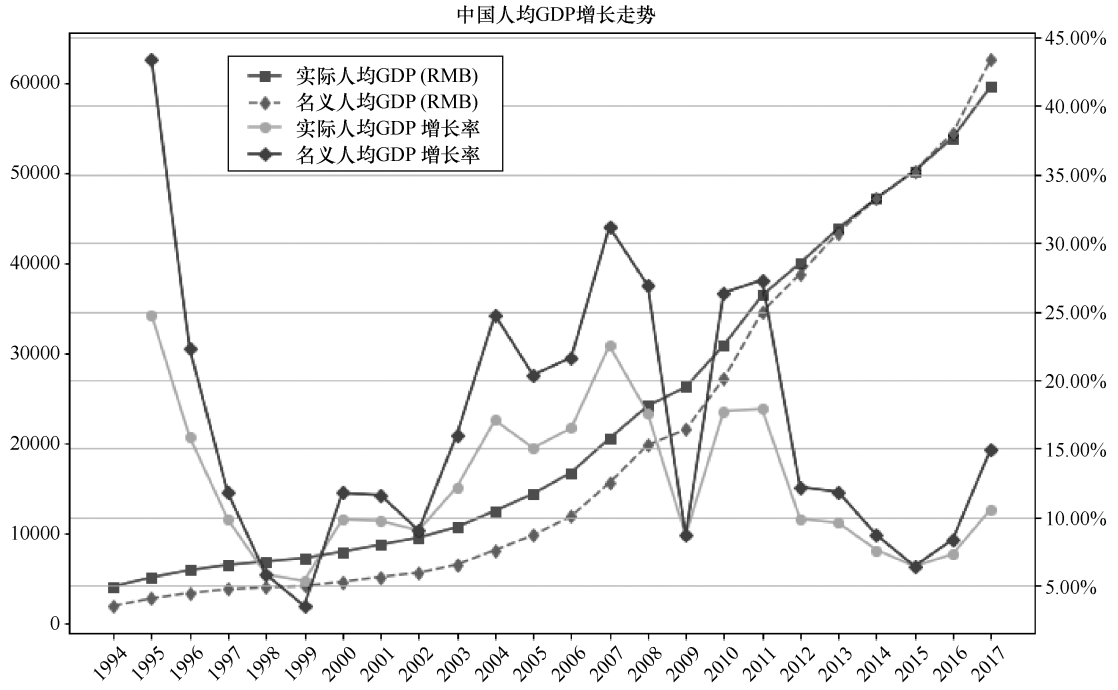


图 11.9 中国人均 GDP 增长走势

计算人均 GDP 增长趋势的完整程序如下：

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import FormatStrFormatter

df = pd.read_csv('gdpchina19942017.csv')
df['pcaprate'] = np.round((df['pcap'] -
                           df['pcap'].shift(1))/df['pcap'].shift(1), 6)*100
df['normpcap'] = np.round(df['pcap']*np.round(df['defl']/100, 2), 4)
df['normpcaprate'] = np.round((df['normpcap'] -
                       df['normpcap'].shift(1))/df['normpcap'].shift(1), 6)*100
ymajorFormatter = FormatStrFormatter('%.2f%%')  # 设置 y 轴标签文本的格式
fig, ax = plt.subplots(figsize = (11, 7))
```

```
plt.rcParams['font.sans-serif'] = ['SimHei']  # 用来正常显示中文标签
plt.xticks(df.index, df.year)
plt.xticks(rotation=50)
ax.plot(df.index, df['pcap'], '-rs', lw = 1.5, label = '实际人均 GDP(RMB)')
ax.plot(df.index, df['normpcap'], '--gd', lw=1.5, label = '名义人均 GDP(RMB)')
plt.legend(loc = 'upper center')
ax2 = ax.twinx()  # 这是双坐标的关键一步
ax2.yaxis.set_major_formatter(ymajorFormatter)  # 显示百分比
ax2.plot(df.index, df['pcaprate'], '-yo', lw = 1.5, label = '实际人均 GDP 增长率')
ax2.plot(df.index, df['normpcaprate'], '-bD', lw = 1.5, label = '名义人均 GDP 增长率')
plt.legend(loc = 'lower center')
plt.title('中国人均 GDP 增长走势')
plt.grid(True)
plt.show()
```

（3）汇率

汇率是指一国货币与另一国货币的比率或比价，或者说是用一国货币表示的另一国货币的价格。实际有效汇率（real effective exchange rate）指数是经本国与所选择国家间的相对价格水平或成本指标调整的名义有效汇率。实际有效汇率指数上升代表本国货币相对价值上升，指数下降则表示本币贬值。当一国的实际有效汇率下降时，意味着该国货币的贬值幅度较之其主要贸易伙伴国货币贬值的幅度更大，该国商品的国际竞争能力相对提高，有利于出口而不利于进口，贸易收支容易出现贸易顺差。反之则容易出现贸易逆差。一个国家的名义有效汇率（nominal effective exchange rate）等于其货币与所有贸易伙伴国货币双边名义汇率的加权平均数，剔除通货膨胀对各国货币购买力的影响就得到实际有效汇率。

国际货币基金组织（IMF）给出的人民币实际有效汇率是以 2000 年平均值为基期 100 进行指数化后的汇率，指数增加表示人民币升值。世界银行公开数据平台也提供了人民币的名义有效汇率数据。这两种有效汇率数据都包含每个月、每个季度和每年的汇率。

图 11.10 是自 2015 年 10 月至 2018 年 6 月间每个月的汇率指标数据的点线图。从图中的汇率曲线看，我国的实际有效汇率和名义有效汇率不仅在数值上差异明显，波动的程度也稍有不同。

如下程序的功能是绘制我国自 2015 年 10 以来的名义有效汇率和实际有效汇率的变化图形（见图 11.10），程序中读取的文件 exchangerate20152018.csv 是经过数据清洗后的有效汇率数据文件。

```
import pandas as pd
import matplotlib.pyplot as plt

gaodf = pd.read_csv('exchangerate20152018.csv')
fig, ax = plt.subplots(figsize = (9, 6))
plt.rcParams['font.sans-serif'] = ['SimHei']  # 用来正常显示中文标签
plt.xticks(gaodf.index, gaodf.Times)
plt.xticks(rotation = 50)
plt.plot(gaodf.index, gaodf.REER, '-rs', lw = 1.5, label = '实际有效汇率')
plt.plot(gaodf.index, gaodf.NEER, '--gd', lw = 1.5, label = '名义有效汇率')
plt.legend(loc = 0)
plt.title('人民有效币汇率')
plt.grid(True)
plt.show()
```

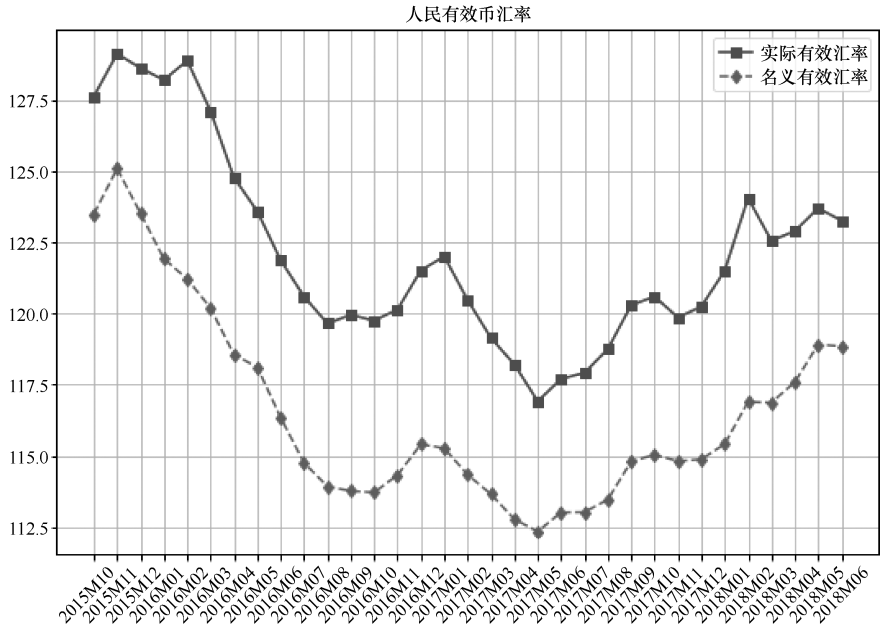


图 11.10　人民币有效汇率

世界银行公开数据平台（https://data.worldbank.org.cn/）提供的数据种类繁多，不同经济数据的统计与发布的时期不完全相同。读者应根据自己的实际需要选择从平台下载合适的数据，根据程序设计的要求对数据进行必要的整理或清洗，并保存为合适的文件类型以供程序使用。

11.5　本章小结

金融数据分析是一项专业交叉性比较强的工作，不仅需要正确理解金融数据的本质含义和相关分析模型的数学含义，而且需要用计算机程序自动实现各种金融分析模型并可视化输出分析的结果。本章介绍了如何用 Python 程序实现股票收益率、波动率、相关系数、MACD 和 KDJ 等常见股票价格分析指标的简单计算与图表输出，通过简单计算和可视化国内生产总值（GDP）和汇率两个宏观经济指标也展示了 Python 在金融数据分析领域应用的价值。

习题

1. 设计一个程序，比较在相同贷款额度（如 100 万元人民币）但不同按揭（月）额度（至少计算三种按揭额度）的情况下各自还款总额的变化趋势。
2. 设计一个程序，计算股票的季收益率和年收益率，并计算单只股票收益率相对同一时期股票市场平均收益率的超额收益率（即相对收益率）。
3. 设计一个程序，利用世界银行公开数据平台提供的宏观经济数据比较最近 40 年间中国与美国两个国家的人均国内生产总值的增长率（图表输出）。
4. 设计一个程序，利用世界银行公开的数据分析我国近 25 年来农村居民与城市居民享受基本公共卫生服务的变化情况（图表输出）。
5. 设计一个程序，计算一年内以 MACD 指标为买入信号、卖出信号进行股票交易而产生的收益。MACD 交易信号为：快线从下往上穿越慢线是当天的买入信号，快线从上往下穿越慢线是当天的卖出信号。假设买入和卖出价格为发出交易信号当天的收盘价格。

第 12 章　配套实验

实验 1　Python 和内置函数

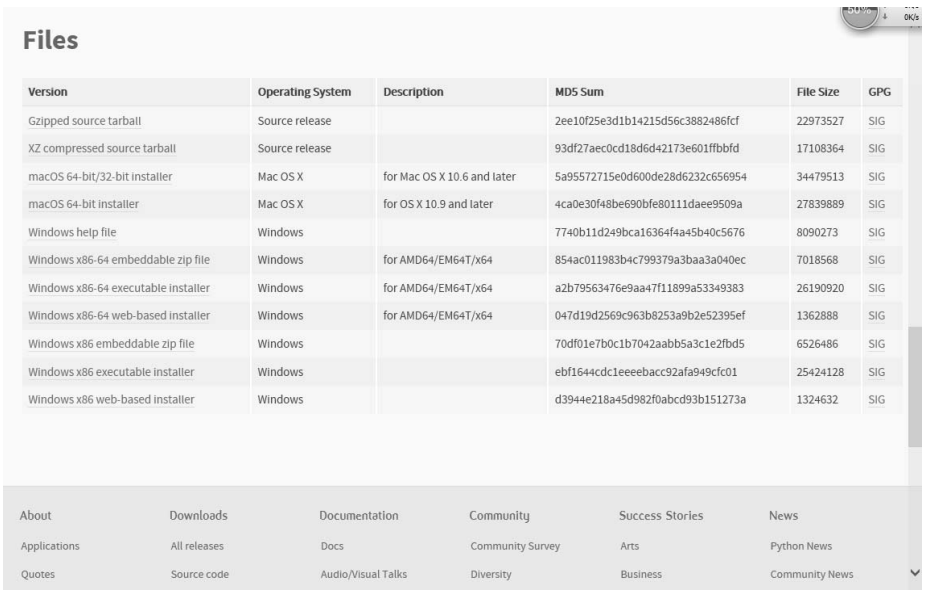
一、实验目的

（1）学习 Python 开发包的下载、安装和 IDLE 的使用。
（2）学习如何在 IDLE 中编辑并运行程序。
（3）学习使用 pip 工具和 conda 工具管理 Python 软件包。
（4）学习 IPython 开发工具的使用。
（5）学习内置函数的使用。

二、实验内容

1．Python 开发包安装与 IDLE 练习

（1）在 Python 的官网下载 Python 安装程序，目前的最新版本为 Python 3.7.3，下载链接为 https://www.python.org/downloads/release/python-373/，在网站页面的 Files 列表中提供了 Windows 和 Mac OS 等操作系统的 Python 开发包，可以根据需要选择下载，如图 12.1 所示。



Files

Version	Operating System	Description	MD5 Sum	File Size	GPG
Gzipped source tarball	Source release		2ee10f25e3d1b14215d56c3882486fcf	22973527	SIG
XZ compressed source tarball	Source release		93df27aec0cd18d6d42173e601ffbbfd	17108364	SIG
macOS 64-bit/32-bit installer	Mac OS X	for Mac OS X 10.6 and later	5a95572715e0d600de28d6232c656954	34479513	SIG
macOS 64-bit installer	Mac OS X	for OS X 10.9 and later	4ca0e30f48be690bfe80111daee9509a	27839889	SIG
Windows help file	Windows		7740b11d249bca16364f4a45b40c5676	8090273	SIG
Windows x86-64 embeddable zip file	Windows	for AMD64/EM64T/x64	854ac011983b4c799379a3baa3a040ec	7018568	SIG
Windows x86-64 executable installer	Windows	for AMD64/EM64T/x64	a2b79563476e9aa47f11899a53349383	26190920	SIG
Windows x86-64 web-based installer	Windows	for AMD64/EM64T/x64	047d19d2569c963b8253a9b2e52395ef	1362888	SIG
Windows x86 embeddable zip file	Windows		70df01e7b0c1b7042aabb5a3c1e2fbd5	6526486	SIG
Windows x86 executable installer	Windows		ebf1644cdc1eeeebacc92afa949cfc01	25424128	SIG
Windows x86 web-based installer	Windows		d3944e218a45d982f0abcd93b151273a	1324632	SIG

About　Downloads　Documentation　Community　Success Stories　News
Applications　All releases　Docs　Community Survey　Arts　Python News
Quotes　Source code　Audio/Visual Talks　Diversity　Business　Community News

图 12.1　Python 安装程序下载页面

（2）鼠标双击已下载的 Python 安装程序进入 Python 安装向导，屏幕上显示如图 12.2 所示 Python 安装方式选择界面。

（3）为了便于在开发中启动各种 Python 工具，建议在图 12.2 中选中 Add Python 3.7 to PATH 复选框。

（4）在图 12.2 中单击 Install Now 链接开始进行系统安装，屏幕上将显示 Python 安装进度界面，如图 12.3 所示。Python 安装完成后，系统显示如图 12.4 所示的 Python 安装成功界面。

（5）在图 12.4 中单击 Close 按钮结束安装过程。

图 12.2　Python 安装方式选择界面

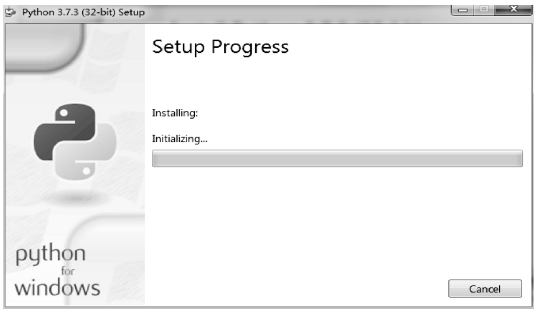


图 12.3　Python 安装进度界面

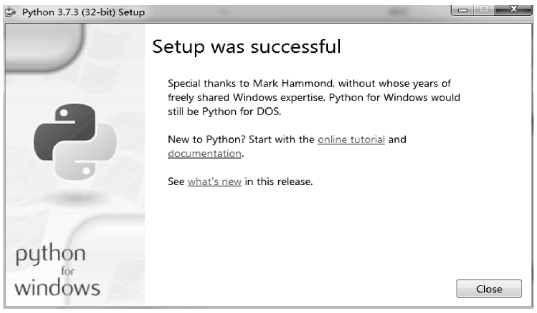


图 12.4　Python 安装成功界面

安装 Python 后，可启动 Python 3.7 程序组中的 IDLE，完成下列基本语句的练习。练习时注意“>>>”是 IDLE 的提示符，输入语句时，语句和提示符之间不能有空格。例如，输入“>>>1+2”将报错，必须输入“>>>1+2”。执行语句中的符号都应是英文字符，不要误输中文标点符号。

```
>>>1+2
>>>10+30.5
>>>10-30.5
>>>10*30.5
>>>x=14; y=4            # 定义两个变量 x，y。在同一行包含两条语句时，中间用分号;隔开
>>>x/y                  # 普通除法，结果为 3.5
>>>x//y                 # //是整除，只保留商的整数部分，结果为 3
>>>2**4                 # **次方运算
>>>3**(1/2)             # 3 开根号
>>>10**1000             # 10 的 1000 次方，表明 Python 可容纳很大的整数
>>>5%2                  # %求余数，余数为 1
>>>4%2                  # 余数为 0
>>>23%5                 # 余数为 3
>>>print(x, y)          # 输出函数
>>>'x={} y={}'.format(x,y)  # 格式化输出，'{}'.format 是一种字符串格式化方法
```

```
>>>x=14; y=4; z=2
>>>x>y                  # True，表示结果为真
>>>x<y                  # False，表示结果为假
>>>x==y                 # 相等比较，注意相等比较是两个=
>>>4==4                 # True 相等
>>>4=='4'               # False 不等，整数 4 和字符串'4'是不等的
>>>x!=y                 # 不相等比较
>>>4+'4'                # 将报错，表明整数和字符串不能相加
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>4+int('4')           # 正确，用 int 函数将字符串转换为整数

>>>abs(-10)             # 绝对值
>>>help(abs)            # 显示 abs 函数的帮助说明

>>>round(3.1415)        # 四舍五入取整
>>>round(3.1415, 2)     # 四舍五入，保留 2 位小数，3.14
>>>round(3.1415, 3)     # 3.142

>>>2**5                 # 2 的 5 次方
>>>pow(2, 5)            # pow 次方函数
>>>pow(2, 1/3)          # 2 开 3 次方
```

2．Anaconda 的下载与安装

（1）下载 Anaconda 安装程序。

（2）双击下载的 Anaconda 安装程序进入 Anaconda 安装向导，显示如图 12.5 所示 Anaconda 安装欢迎界面。

（3）单击界面上的 Next 按钮进入 Anaconda 权限许可界面，如图 12.6 所示。

图 12.5　Anaconda 安装欢迎界面

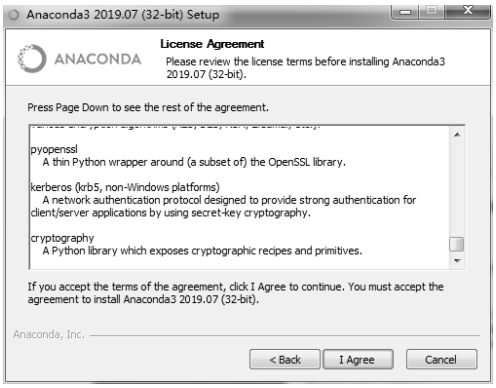


图 12.6　Anaconda 权限许可界面

（4）单击图 12.6 中的 I Agree 按钮进入如图 12.7 所示的 Anaconda 安装类型选择界面，默认的选项为 All User (requires admin privileges)。

（5）单击图 12.7 中的 Next 按钮进入 Anaconda 安装位置设定界面，如图 12.8 所示。Anaconda 默认的安装位置为 C:\ProgramData\Anaconda3。

（6）设定好目录位置后，单击 Next 按钮开始安装 Anaconda，按照安装向导提示一直单击 Next 按钮，直到出现如图 12.9 所示的 Anaconda 安装结束界面，单击界面上的 Finish 按钮结束安装过程。

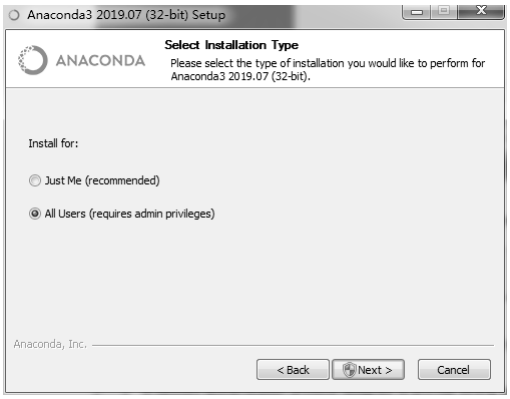


图 12.7 Anaconda 安装类型选择界面

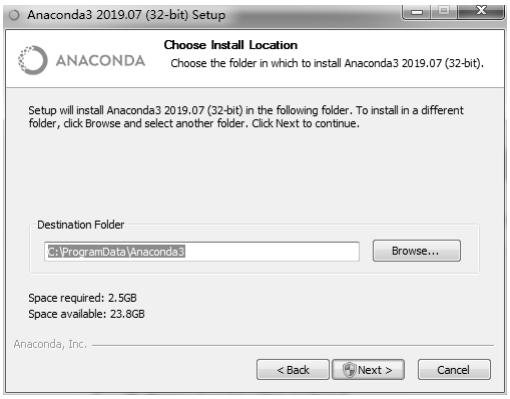


图 12.8 Anaconda 安装位置设定界面

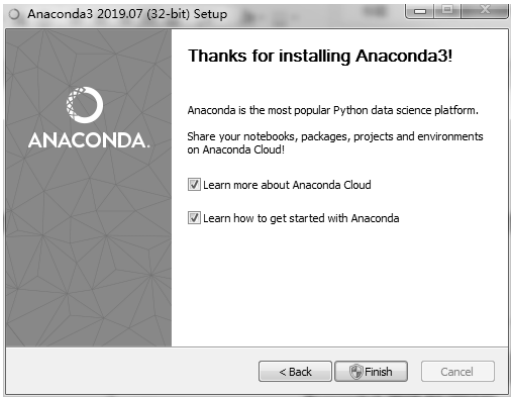


图 12.9 Anaconda 安装结束界面

3．pip 工具和 conda 工具

（1）启动 Anaconda Prompt 窗口

Anaconda 软件中包含了 pip 工具和 conda 工具，因此安装 Anaconda 软件后就能使用 pip 和 conda。pip 和 conda 可在 Anaconda Prompt 窗口使用，启动 Anaconda Prompt 窗口的方式如下。

选择 Windows 的“开始”→“所有程序”→Anaconda3→Anaconda Prompt 命令，打开如图 12.10 所示的 Anaconda Prompt 窗口。

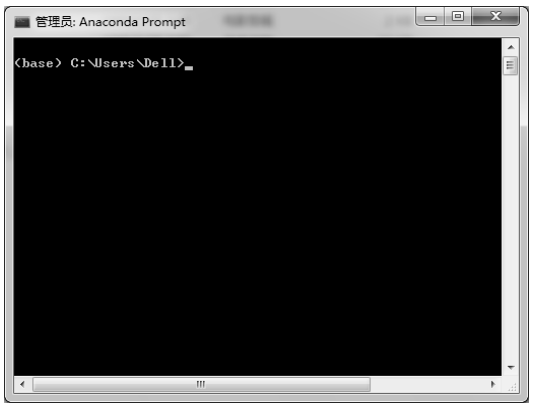


图 12.10 Anaconda Prompt 窗口

（2）管理软件包

在 Anaconda Prompt 窗口中分别输入如下命令实现对软件包的管理操作。

① 输入“pip list”或“conda list”命令，查看 Python 已安装的所有软件包。

② 输入“pip list --outdated”或“conda list --outdated”命令，查看哪些已安装的软件包需要升级。

③ 输入“pip install --upgrade 软件包名”或“conda install --upgrade 软件包名”命令，更新已安装的 Python 软件包。

④ 输入“pip install 软件包名”或“conda install 软件包名”命令，安装 Python 软件包。示例如下：

```
pip  install  pillow          # 安装 pillow 图片处理库
pip  install  jieba           # 安装 jieba 中文分词库
pip  install  python-docx     # 安装 python-docx 包以读取 Word 文档
```

⑤ 输入“pip uninstall 软件包名”或“conda uninstall 软件包名”命令，卸载已安装的 Python 软件包。

4．IPython 开发工具的使用

（1）启动 IPython 交互式命令窗口可以采用如下两种方法。

方法 1：启动 Anaconda Prompt 窗口，在窗口中输入“ipython”命令就会进入 IPython 交互式命令窗口，如图 12.11 所示。要退出 IPython，可执行 quit()命令。

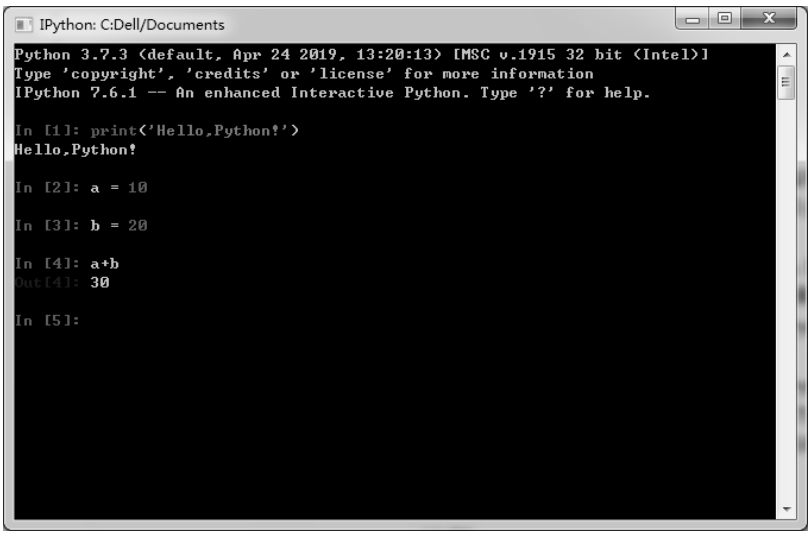


图 12.11　IPython 交互式命令窗口

方法 2：选择 Windows 的“开始”→“所有程序”→Anaconda3→Spyder 命令，打开如图 12.12 所示的 Spyder 集成开发环境窗口。Spyder 开发环境提供了 IPython 交互命令窗格。

（2）在 IPython 交互式命令窗口的“In [序号]:”提示符后面输入如下语句，查看程序执行结果。

```
In: print('Hello,Python')
In: x = 10
In: print(x)
```

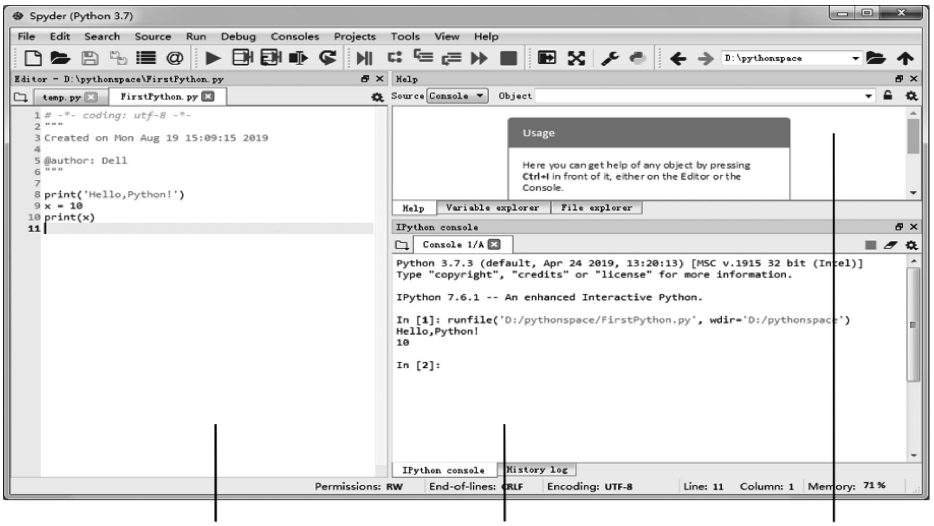


图 12.12　Spyder 集成开发环境窗口

5．运算符和表达式

（1）写出下列公式的 Python 表达式

① $5+(2\times3)^2$

```
In: 5+(2*3)**2
```

② $2a(b+5)$　　# 设定 $a=2; b=3$

```
In: a=2; b=3
In: 2*a*(b+5)
```

③ $\cos^2(a+b)$

```
In: import math
In: math.cos(a+b)**2
```

（2）先手工计算下列表达式的值，然后在 IPython 窗口中验证。

① `1 + 2**3 + 4 % 5 + (6 > 7) + 8 * 9`

② `not a <= c or 4 * c == b**2 and b != a + c`　　`# 设定 a=2; b=3; c=4`

③ `3 > a * b or a == c and b != c or c > d`　　`# 设定 a=2; b=3; c=4; d=5`

6．内置函数的使用

（1）在 IPython 命令窗口中实现：输入一个字符串，统计字符串中的字符个数、最大字符、最小字符，将字符串中的所有字母大写、所有字母小写、所有字母大小写互换、每个单词的首字母大写，查找字符串中特定字符（如 Python）的位置。

分析：使用 input()函数可以读取键盘输入的内容，函数 len()、max()、min()分别统计字符串中字符的个数、最大字符、最小字符，函数 upper()将字符串中的所有字母转为大写，函数 lower()将字符串中的所有字母转为小写，函数 swapcase()将字符串中的所有字母大小写互换，函数 title()将每个单词的首字母大写，字符串查找函数为 find()或 index()。程序如下所示：

```
In: s = input('输入一串字符：')
输入一串字符：Hi,This is the first Python program
In: len(s)                                    # 统计字符个数
Out: 35
```

```
In: max(s)                                          # 最大的字符，按 ASCII 码比较大小
Out: 'y'
In: min(s)                                          # 最小的字符
Out: ' '
In: ord('y'), ord(' ')                              # 显示'y'和空格符的 ASCII 码
Out: (121, 32)
In: s.upper()                                       # 所有字母大写
Out: 'HI,THIS IS THE FIRST PYTHON PROGRAM'
In: s.lower()                                       # 所有字母小写
Out: 'hi,this is the first python program'
In: s.swapcase()                                    # 改变字母的大小写
Out: 'hI,tHIS IS THE FIRST pYTHON PROGRAM'
In: s.title()                                       # 每个单词首字母大写，其他小写
Out: 'Hi,This Is The First Python Program'
In: s.find('Python')                                # .find()查找单词 Python 的位置
Out: 21
In: s.find('abc')                                   # 查找'abc'位置，未找到返回-1
Out: -1
In: s.index('Python')                               # .index()也可查找
Out: 21
In: s.index('abc')                                  # .index()查找不到将报错
ValueError: substring not found
```

（2）在 IPython 命令窗口中实现：根据圆的半径计算圆的周长和面积。

分析：在计算圆的周长和面积时，需要使用 math 模块中的 pi 常量，因此程序需要导入 math 模块。在输出结果时，使用 format 进行格式化输出。程序代码及运行结果如下：

```
In: import math                                                          # 导入 math 模块
In: r = 2.5
In: perimeter = 2 * math.pi * r                                          # 计算周长
In: print('圆的半径为：{:.2f},周长为：{:.2f}'.format(r, perimeter)) # 格式化输出周长
圆的半径为：2.50,周长为：15.71
In: area = math.pi * r * r                                               # 计算面积
In: print('圆的半径为：{:.2f},面积为：{:.2f}'.format(r, area))    # 格式化输出面积
圆的半径为：2.50,面积为：19.63
```

（3）在 IPython 命令窗口中实现：输入三角形的三条边，计算三角形的面积。

分析：程序需要使用 math 模块中的 sqrt()函数开平方，因此程序需要导入 math 模块。使用 input()函数获得的数据为字符串类型，需要使用 eval()函数将输入的数据转换为数值型数据。在输出计算结果时，使用 format 进行格式化输出。程序代码及运行结果如下：

```
In: import math                              # 导入 math 模块
In: a = input('输入第一条边的长度：')
输入第一条边的长度：3
In: b = input('输入第二条边的长度：')
输入第二条边的长度：4
In: c = input('输入第三条边的长度：')
输入第三条边的长度：5
In: a = eval(a)                              # eval()函数将字符串转换为数值
In: b = eval(b)
In: c = eval(c)
```

```
In: p = (a + b + c) / 2                    # 根据海伦公式计算面积
In: area = math.sqrt(p*(p-a)*(p-b)*(p-c)) # math.sqrt()平方根函数
In : print('三角形的面积为：{:.2f}'.format(area))
三角形的面积为：6.00
```

单条语句调试成功后，将上述语句输入到 Spyder 的左侧源程序窗口中，保存为 area.py 程序文件。单击工具栏上的绿色三角形“运行”按钮执行该程序，如图 12.13 所示。执行后查看 Spyder 右上角的 Variable explorer 变量显示窗格，观察窗格中的变量情况。

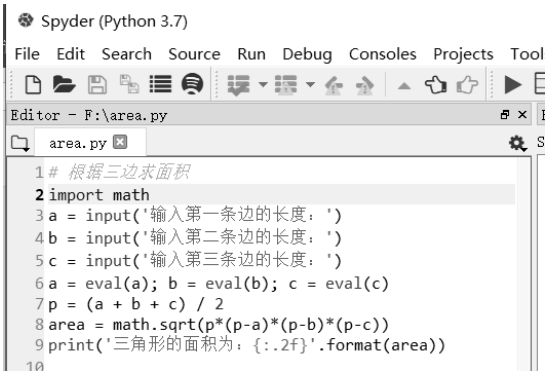


图 12.13　面积计算程序

实验 2　列表、元组、字典和集合

一、实验目的

（1）了解列表、元组、字典和集合的概念。
（2）学习列表、元组、字典和集合对象的创建。
（3）学习列表、元组、字典和集合函数的使用。
（4）了解浅复制和深复制的区别。

二、实验内容

1．对照教材 3.1 节练习列表的各种基本操作。
2．列表切片练习。
假设有一个列表 `lst = [1,2,3,4,5,6,7,8,9]`，分别进行下面的切片操作：
（1）取前 3 个元素。
（2）从索引 2 开始，连续取 3 个元素。
（3）取索引为奇数的元素。
（4）取最右侧的 4 个元素。
分析：序列类型的元素索引分为正索引和负索引两种情况。正索引的索引值从 0 开始，从列表最左向右依次递增 1。负索引的索引值从-1 开始，从列表最右向左依次递减 1。在列表的切片操作中有三个参数：start 表示列表切片的开始索引位置，end 表示列表切片的终止索引位置，

step 表示步长。程序代码如下：

```
In: lst = [1, 2, 3, 4, 5, 6, 7, 8, 9]      # 两种方式均可得到所需列表
In: lst = list(range(1,10))
In: lst[:3]                                # 取前 3 个元素，也可写为 lst[0:3]
Out: [1, 2, 3]
In: lst[2:5]                               # 从索引 2 开始，连续取 3 个元素
Out: [3, 4, 5]
In: lst[1:9:2]                             # 取索引为奇数的元素
Out: [2, 4, 6, 8]
In: lst[-4:]                               # 取最右侧的 4 个元素
Out: [6, 7, 8, 9]
```

3．统计列表 lst = [12,2,16,30,28,10,16,20,6,18]中元素的个数，找出最大值和最小值，计算平均值，并将 lst 列表按从大到小排序输出。

分析：列表对象使用[]符号创建，使用 len()函数统计列表中元素的个数，max()函数获得列表中的最大值元素，min()函数获得最小值元素。使用 sort()函数可以对列表中的元素排序，默认的排序方式是从小到大。要实现从大到小的逆序排序，需要在 sort()函数排序时设置参数 reverse=True。程序代码如下：

```
In: lst = [12, 2, 16, 30, 28, 10, 16, 20, 6, 18]
In: len(lst)                               # 元素个数
Out: 10
In: max(lst)                               # 最大值
Out: 30
In: min(lst)                               # 最小值
Out: 2
In: lst.sort(reverse=True)                 # reverse=True 表示进行逆排序
In: lst
Out: [30, 28, 20, 18, 16, 16, 12, 10, 6, 2]
In: sum(lst)                               # 求和
Out: 158
In: sum(lst) / len(lst)                    # 求平均值
Out: 15.8
```

4．使用字典保存用户姓名和对应密码，输出所有用户姓名，并找出某个用户的密码。

分析：字典对象使用{}符号创建。用户姓名和对应密码保存为字典数据类型，姓名为字典中的“键”，密码则为“值”。使用字典中的 keys()函数可以获得所有的“键”，使用 get()函数可以获得字典中某个“键”对应的“值”。程序代码如下：

```
In: password = {'张三': '123456', '李四': 'abcdef', '王五': '123abc', '刘六': 'aaa111'}
In: password.keys()
Out: dict_keys(['张三', '李四', '王五', '刘六'])
In: password.get('李四')
Out: 'abcdef'
In: for  k, v  in  password.items():          # 遍历输出所有的键值对
        print(k, v)
```

5．使用元组分别保存学生姓名和对应成绩，找出成绩最高分学生的姓名。

分析：假设学生姓名和成绩的元组分别为 student 和 score，首先在 score 中使用 max()函数找出最高成绩，然后用 index()函数获得最高成绩所在的索引位置，那么 student 元组对应的

索引位置即为该学生的姓名。程序代码如下：

```
In: student = ('张三', '李四', '王五', '刘六')
In: score = (75, 62, 93, 81)
In: m = max(score)              # 在 score 中找出最高分
In: idx = score.index(m)        # 获得最高分在元组中的索引值
In: idx
Out: 2
In: '{}同学的成绩{}是最高成绩'.format(student[idx], m)
Out: '王五同学的成绩 93 是最高成绩'
```

6．输入一串字符，统计单词个数（假设所有单词以空格分隔）。

分析：由于所有单词以空格分隔，因此可以首先使用字符串的 split()函数将字符串按空格分隔得到列表，然后使用 len()函数返回列表长度。程序代码如下：

```
In: s = input('输入一串字符：')
输入一串字符：this is a Python  program
In: words = s.split()                                    # 默认以空格分隔字符串
In: words                                                # 获得单词列表
Out: ['this', 'is', 'a', 'Python', 'program']
In: '字符串中单词个数为：{}'.format(len(words))
Out: 字符串中单词个数为：5
```

7．合并列表 lst1 = [3,7,44,78,6]和 lst2 = [35,8,59,3,47,6]中的元素，并将重复元素去除。

分析：首先由列表 lst1 的元素创建一个集合，然后使用集合中的 update()函数添加 lst2 的元素。由于集合中不存在重复元素，将添加元素后的集合再转换为列表就完成了去重操作。程序代码如下：

```
In: lst1 = [3, 7, 44, 78, 6]
In: lst2 = [35, 8, 59, 3, 47, 6]
In: s = set(lst1)                    # 由 lst1 创建集合
In: s.update(lst2)                   # 将 lst2 添加到集合中
In: s
Out: {3, 6, 7, 8, 35, 44, 47, 59, 78}
In: list(s)                          # 由集合得到合并后的列表
Out: [3, 35, 6, 7, 8, 44, 78, 47, 59]
In: list(set(lst1 + lst2))           # 上述步骤也可以合并为一条语句完成
Out: [3, 35, 6, 7, 8, 44, 78, 47, 59]
```

8．浅复制和深复制。

分析：对可变类型的浅复制会开辟新的空间地址。浅复制后，如果改变原始对象中可变类型元素的值，那么会同时改变复制对象的值。对可变类型来说，深复制后两个对象的内存地址会彻底分开，两个对象将完全独立。程序代码如下。

（1）浅复制

```
In: import copy                                  # 导入 copy 模块
In: lst1 = ['a', 'b', 'c', [1, 2, 3]]
In: lst2 = lst1.copy()                           # 浅复制
In: lst2
Out: ['a', 'b', 'c', [1, 2, 3]]                  # 复制后 lst2 的元素
In: id(lst1), id(lst2)
```

```
Out: (410766456    , 214144784)                    # lst1 和 lst2 的内存地址不同
In: id(lst1[3]), id(lst2[3])
Out: (758376523, 758376523)        # 浅复制后两个列表中的可变元素的内存地址相同
In: lst1[3].append(4)                              # 对 lst1 中的列表元素添加元素
In: lst1
Out: ['a', 'b', 'c', [1, 2, 3, 4]]                 # 添加元素后的 lst1 列表
In: lst2
Out: ['a', 'b', 'c', [1, 2, 3, 4]]                 # lst2 中的列表元素也受影响
```

（2）深复制

```
In: import copy                                    # 导入 copy 模块
In: lst1 = ['a', 'b', 'c', [1, 2, 3]]
In: lst2 = copy.deepcopy(lst1)                     # deepcopy()深复制
In: lst2
Out: ['a', 'b', 'c', [1, 2, 3]]                    # 复制后 lst2 的元素
In: id(lst1), id(lst2)
Out: (176087968, 214210520)                        # lst1 和 lst2 的内存地址不同
In: id(lst1[3]), id(lst2[3])
Out: (378128374, 457837583)      # 深复制后两个列表中的可变元素的内存地址也不相同
In: lst1[3].append(4)                              # 对 lst1 中的列表元素添加元素
In: lst1
Out: ['a', 'b', 'c', [1, 2, 3, 4]]                 # 添加元素后的 lst1 列表
In: lst2
Out: ['a', 'b', 'c', [1, 2, 3]]                    # lst2 中的列表元素未受影响
```

实验 3　程序的流程控制

一、实验目的

（1）掌握用 if 语句编写选择结构程序。

（2）掌握用 while 语句编写循环结构程序。

（3）掌握用 for 语句编写循环结构程序。

（4）掌握用 continue 和 break 语句实现流程跳转。

二、实验内容

1．输入一个日期，包括年、月和日，计算该日期是这一年的第几天。

2．输入三个整数，把这三个整数按从大到小的顺序输出。

3．分别用 while 语句和 for 语句编写程序，计算整数 100 到 1000 的和。

4．随机生成两个 1000 到 10000 之间的整数，若这两个整数之间有素数，则输出最小的一个，否则什么都不输出。

5．国际象棋棋盘共有 64 个方格，现在第一个格子上放 1 颗麦粒，以后每一个格子都比前一个格子的麦粒数翻倍。计算放满整个棋盘需要的麦粒总数。现设 1 颗麦粒重 50 毫克，小麦共重

多少吨？

6．编程输出九九乘法表。

实验 4　函数练习

一、实验目的

（1）了解 Python 函数的概念。

（2）掌握 Python 函数设计。

（3）熟练使用函数调用的各种方式。

（4）学会编写 lambda 表达式。

（5）学会编写和使用生成器对象。

（6）掌握 turtle、pyinstaller 库的使用方法。

二、实验内容

1．定义一个无参函数，输出“欢迎您的到来!”，然后在主程序中调用该函数。

2．定义一个函数，参数为一个实数，代表一个摄氏温度值，将它转换为一个华氏温度值，并返回该值（$F = 1.8C + 32$）。在主程序中调用该函数。

3．定义一个函数，函数参数为一个小于 10000 的正整数，分解它的各位数字，并以一个元组的形式返回。在主程序中调用该函数。

4．编写一个 lambda 表达式，对给定的列表[1, 2, 3, 4, 5]，把它的每个元素值分别加上 10，生成一个新列表。

5．定义一个 yield 生成器函数，生成 200 以下 5 的所有倍数。

6．练习随机数应用。请生成 50 个随机数据，模拟一个班的考试成绩（要求在 40～100 分之间）。计算这批数据的平均分、最高分和最低分，并排序由高到低输出。

7．使用 turtle 库绘制一个五角星图案。示例代码如下：

```
import turtle
turtle.color('yellow', 'red')    # 设置线条黄色，填充色红色
turtle.begin_fill()              # 开始填充
for  _  in  range(5):            # 五角星有 5 条线
    turtle.forward(200)          # 前进 200 像素，即边长 200
    turtle.right(144)            # 右转 144 度
turtle.end_fill()                # 结束填充
turtle.done()                    # 结束绘图，见图 12.14
```

8．将第 7 题的代码改写为一个绘制五角星的函数，可以绘制指定边长和填充色的五角星。在画板上随机绘制若干五角星。示例代码如下：

```
import turtle as t
import random
def drawstar(line,color):              # 绘制五角星函数
    t.fillcolor(color)
```

```
        t.begin_fill()
        for _  in  range(5):              # 5 条线
            t.forward(line)               # 前进 line
            t.right(144)                  # 右转 144 度
        t.end_fill()
    def gotopos(x,y):                     # 函数，用于将绘图乌龟移到坐标(x,y)
        t.penup()                         # 抬起画笔
        t.setpos(x,y)                     # 移到指定坐标(x,y)
        t.pendown()                       # 放下画笔

    t.speed(10)                           # 主程序开始，设置绘图速度
    colors = ['red', 'green', 'blue', 'orange', 'yellow', 'pink']
    for  _   in  range(20):               # 绘制 20 个五角星
        x = random.randint(-200, 200)     # 随机坐标点(x,y)
        y = random.randint(-200, 200)
        line = random.randint(10, 100)    # 随机线长
        color = random.choice(colors)     # 随机选一种颜色
        gotopos(x, y)                     # 移动到(x,y)上
        drawstar(line, color)             # 绘制五角星
    t.done()                              # 见图 12.15
```

图 12.14　turtle 库绘制五角星

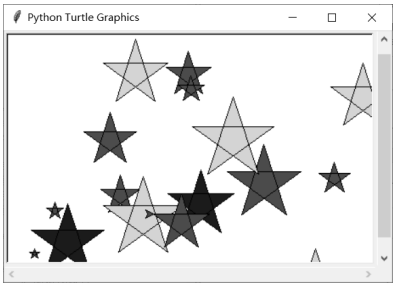


图 12.15　随机位置绘制五角星

9．利用 pyinstaller 库将第 8 题保存的程序文件 p8.py 打包为一个可执行程序。

步骤 1：进入 Windows 命令行，执行下面的命令安装 pyinstaller 库：

```
pip install pyinstaller
```

步骤 2：在命令行窗口，切换到 p8.py 所在的文件夹，执行如下打包命令：

```
pyinstaller -F -w p8.py
```

命令执行后，在当前目录下会产生一个 dist 子目录，子目录中含有生成的 p8.exe 可执行程序，双击即可运行。

实验 5　文件读写

一、实验目的

（1）了解文件和文件对象的概念。

（2）掌握文本文件的读写方法。

（3）掌握二进制文件的读写方法。

（4）掌握 docx 文件和 xlsx 文件的读写方法。

（5）掌握 jieba、wordcloud 第三方库的使用。

二、实验内容

1．将整数 12345 分别写入文本文件 test.txt 和二进制文件 test.dat，并比较两个文件的不同之处。

2．以只写方式打开一个文本文件 file.txt，将一个实数写入该文件，然后关闭文件。用只读方式打开同样的文件，读入一个实数，然后关闭该文件。最后输出读入的实数。

3．当前目录下有一个文本文件 data.txt（该文件需自行构造），文件的内容为若干件商品一月到六月的销售额，各月销售额之间用英文逗号分隔，查找并输出 6 个月总销售额最小的商品及其总销售额。

4．当前目录下有一个 Word 文档，读取该文件内容并统计其中每个字出现的频次。

5．当前目录下有一个 xlsx 工作簿，第一列为学生姓名，第二列到第四列分别是学生的语文、数学和英语成绩。编写程序计算每名学生的总分，并输出成绩前三名的学生姓名。

6．从网上下载一篇英文小说，统计小说中每个单词的出现频次，输出频次最高的 10 个单词。

7．从股票软件中导出某只股票的历史交易数据，计算收盘价的 5 日均价。5 日均价是股票术语：5 日均价 = 最近连续 5 个交易日收盘价之和/5。

8．搜索一篇有关“电子烟”的中文新闻报道，将网页内容复制到 smoker.txt 文本文件中。利用 jieba 库对新闻报道做分词处理，并做出词云图（注意：首先要执行如下两个命令安装需要的库）。

```
pip  install  jieba          # 安装 jieba 分词库
pip  install  wordcloud      # 安装词云库
```

参考代码如下：

```
from   wordcloud  import  WordCloud       # 词云库
import  jieba                             # 分词库
def  wc(note, fname, bgcolor="white"):    # 产生词云图的函数
    wc = WordCloud(background_color=bgcolor, font_path='simfang.ttf') # 背景色，仿宋体
    wc.generate(note)         # 生成词云图片
    wc.to_file(fname)         # 保存为文件

with open('smoker.txt') as f:         # 打开保存的新闻文件 smoker.txt
    s = f.read()
# 这篇文章是关于“电子烟”的新闻报道，向 jieba 库中临时添加新词“电子烟”
# 如不添加新词，“电子烟”将被拆分为“电子”和“烟”两个词
jieba.add_word('电子烟')
wd = jieba.lcut(s)                                  # 分词，得到词列表
words = [x  for  x  in  wd  if  len(x)>1]      # 剔除空格、单字
s = ' '.join(words)     # 将词列表连接为空格分隔的字符串（' '中间必须有一个空格）
wc(s, 'wc.png')         # 调用函数，在当前目录中生成词云图片 wc.png，见图 12.16
```

图 12.16　“电子烟”新闻报道词云图

生成词云后，在 IPython 交互窗口继续探索研究得到的分词列表。

```
In: len(wd), len(words)              # 初始分词的词汇数、剔除了空格和单字后的词汇数
Out: (228, 132)
In: from collections import Counter  # 引入统计函数 Counter
In: dct=Counter(words)               # 对 words 做词频统计，返回类似字典的统计结果
In: dct.most_common(5)               # 输出词频最高的 5 个词
Out: [('电子烟', 7), ('美国', 5), ('病例', 3), ('这些', 2), ('四分之三', 2)]
```

实验 6　NumPy 科学计算库

一、实验目的

（1）安装科学计算库 NumPy。
（2）了解 numpy.ndarray 数组的各种基本属性。
（3）掌握数组的各种访问方式及数组的组合、拆分方法。
（4）掌握常用的统计函数及读取数据文件的方法。

二、实验内容

在 Windows 命令行上执行如下命令安装或升级 NumPy。

```
pip  install  numpy          # 安装 NumPy
pip  install  -U  numpy      # 升级 NumPy
```

启动Spyder，在IPython交互环境下完成下列练习。

1．了解 NumPy 数组的基本属性

```
In: import   numpy  as  np
In: np.__version__                          # 显示 NumPy 版本，注意前后都是两个下画线
In: a = np.arange(15).reshape(3, 5)         # 创建二维数组 a
In: a
In: type(a)                                 # 显示 a 的数据类型，numpy.ndarray
In: a.shape                                 # a 的形状(3,5)
In: a.ndim                                  # 维度为 2
In: a.itemsize                              # 每个数据所需的存储空间 4 字节
```

```
In: a.size                                  # 数据个数 15
In: a.dtype                                 # 数据元素的类型'int32'
```

2．创建 NumPy 数组的各种方法

```
In: b1 = np.array([1, 3, 5, 7, 9])
In: b1.dtype                                # 默认类型 'int32'
In: b2 = np.array([1, 3, 5, 7, 9], dtype='float')  # 指定数据类型 float
In: b2.dtype

In: np.arange(1, 11, 2)
In: np.random.randn(3, 4)                   # 符合 N(0,1)分布的 3x4 数组
In: np.random.randint(1, 100, 5)            # 在区间[1,100]内生成 5 个随机整数
In: np.random.randint(1, 100, (5, 3))       # 生成 5x3 数组，值在区间[1,100]内
In: np.random.uniform(1, 3, size=(3, 4))    # 生成 3x4 随机小数数组，值在区间[1,3)内
In: np.array([(1, 2, 3), (4, 5, 6)])
In: b3 = np.arange(1, 7).reshape(2, 3)
In: np.zeros((3, 4))                        # 全 0 数组，注意(3,4)有小括号
In: np.ones((3, 4))                         # 全 1 数组
In: np.eye(5)                               # 单位矩阵
In: np.full((3, 4), 1.5)                    # 全为 1.5
In: np.tile(b3, 2)                          # 重复填充

In: np.linspace(1, 2, 10)                   # 在区间[1,2]内等间距生成 10 个点，含终值 2
In: np.linspace(1, 2, 10, endpoint=False)   # 在区间[1,2]内等间距生成 10 个点，不含终值 2
In: np.linspace(-2*np.pi, 2*np.pi, 100)
```

3．索引访问 NumPy 数组的各种方法

```
In: a = np.arange(10, 20)             # 一维数组访问练习
In: a[0] ,  a[-1],  a[2:5], a[[1, 4, 5]]
In: a[-3:] , a[::2], a[::-1]
In: a>15
In: a[a>15]                           # 布尔数组筛选

In: b = np.arange(24).reshape(4, 6)   # 二维数组访问练习
In: b[1]
In: b[1, 2]
In: b[:,2]
In: b[1:, 1:3]
In: b[:,::2]

In: c = b[1]                          # c 是 b[1]的视图
In: c[0] = 100
In: b[1, 0]                           # 可见 b[1,0]的值为 100，表明其也被修改
In: d = b[2].copy()                   # d 是 b[2]的复制
In: d[0] = 200
In: b[2, 0]                           # b[2,0]的值不会被修改
```

4．数组基本运算

```
In: a = np.arange(1, 6)
```

```
In: a + 5,  a * 2,  a / 2,  10 - a
In: b = np.arange(10, 15)
In: a + b, a * b,  a / b

In: c = np.arange(15).reshape(3, 5)
In: a + c,  c - a                # 广播运算
```

5．统计函数

```
In: np.random.seed(7)
In: a = np.random.rand(3, 4)
In: np.mean(a),  a.mean()            # 均值
In: np.mean(a, axis=0),  a.mean(axis=0)
In: np.mean(a, axis=1)
In: np.var(a)  ,  np.std(a)          # 方差、标准差
In: np.sin(a),  np.round(a, 2)
In: a[1,::2] = np.nan                # 设置 nan 值
In: np.isnan(a) , np.isnan(a).sum()  # 判断 nan 值
In: np.sum(a),  np.nansum(a)
```

6．排序练习

```
In: np.random.seed(7)                # 设置随机数种子 7
In: b = np.random.randint(1, 20, size=10)
In: c = b.copy()                     # 将数组 b 复制一份，用于后续的测试
In: b.sort()                         # 排序后直接改变了 b 的数据顺序
In: np.sort(c)                       # 返回新的有序数组，不改变 c
In: np.argmax(c), np.argmin(c)       # 返回 c 中最大值、最小值的索引序号
In: np.argsort(c)                    # 返回按顺序排列时各位上取值对应的索引号
In: c[np.argsort(-c)]                # 逆序排列
```

7．组合和拆分数组

```
In: a = np.arange(24).reshape(4, 6)
In: h1, h2 = np.hsplit(a, 2)         # 水平拆分
In: v1, v2 = np.vsplit(a, 2)         # 竖直方向拆分
In: np.vstack((v1, v2))              # 组合
In: np.hstack((h1, h2))
```

8．读写文件

```
In: b = np.arange(12)
In: b.shape = (3, 4)
In: np.savetxt('data.txt', b, delimiter=',' , fmt="%i") # 存为文本文件
In: c = np.loadtxt('data.txt', delimiter=',')           # 读取文本文件
In: b == c                                              # 测试数组 b 和 c 是否相等
```

9．练习

（1）先创建一个 9×9 的全 1 二维数组，再将二维数组四条边上的数据都修改为 0。

（2）在区间[1, 6]内生成 1000 个随机整数，统计每个整数出现的次数。

（3）练习数组的插入（insert）和删除（delete）。

（4）生成 2×3 和 3×2 两个数组，利用 np.dot()计算数组的矩阵乘法。

（5）模拟生成 50 名同学的单科成绩，符合正态分布 $N(70, 100)$。完成排序，求最大值、最小值和均值。

实验 7　Pandas 数据分析库

一、实验目的

（1）学习数据分析库 Pandas 的安装。
（2）掌握 Pandas 中的 Series 和 DataFrame 数据结构。
（3）理解 Pandas 的索引，了解 Pandas 和 NumPy 的区别。
（4）掌握数据框的各种常用操作，掌握分组函数和时间序列函数。

二、实验内容

在 Windows 命令行上执行如下命令安装或升级 Pandas。

```
pip  install  pandas          # 安装 Pandas
pip  install  -U  pandas      # 升级 Pandas
```

启动 Spyder，在 IPython 交互环境下完成下列练习。

1．Series 基本练习

```
In: import  pandas  as  pd
In: from    pandas  import  DataFrame,  Series
In: pd.__version__                          # 显示 Pandas 版本
In: s = Series(np.arange(6), index=list('abcdef'))
In: s.index                                 # 显示索引
In: s.values                                # 显示数据
In: s['a'],  s[0],  s['a':'c'],  s[0:2]     # 索引访问
In: s['c':'e']=3                            # 赋值修改
In: s.sum(), s.mean(), s.median()           # 统计函数
In: s.value_counts()                        # 统计数据的频次
In: s.unique()                              # 返回不重复的数据列
In: s.head(3)                               # 显示头 3 个数据
In: s.tail(2)                               # 显示末尾 2 个数据

In: s2 = Series(np.arange(12, 30, 3), index=list('abcabc'))
In: s + s2                                  # 按索引对齐运算
In: s * s2
```

2．创建和访问数据框 DataFrame

```
In: data = {'apple':[1100, 1050, 1200], 'huawei':[1250, 1300, 1328], 'oppo':[800, 850, 750]}
In: df = DataFrame(data, index=['一月', '二月', '三月'])          # index 标签
In: df.apple,  df.loc['一月']
In: df.iloc[0],  df.iloc[:, 1]
```

```
In: df.at['二月', 'oppo'],  df.iat[1, 2]
In: df.query('huawei>1300')
In: df.index,  df.columns
In: df.to_csv('m5.csv', encoding='cp936')                          # 保存文件
In: df2 = pd.read_csv('m5.csv', index_col=0, encoding='cp936')     # 读取文件
```

3．nan 缺失值处理

构造一个含有 nan 值的 Series，练习 isnull()、notnull()、dropna()、fillna()等函数。

4．统计函数

构造数据框，练习 sum()、count()、mean()、median()、max()、min()、var()等函数。

5．读写文件

（1）构造一个文本数据文件，练习 read_csv()、to_csv()函数。

（2）构造一个 Excel 文件，练习 read_excel()、to_excel()函数。

6．数据整理

（1）练习数据框的 drop()、pop()、reindex()、reset_index()、set_index()、append()方法。

（2）构造含有重复值的数据框，练习针对重复值的 duplicated()、drop_duplicates()方法。

（3）构造数据框，练习 sort_values()和 sort_index()排序方法。

（4）构造两个数据框，含有共同的列，练习 merge 的各种连接方法。

（5）构造区间[0, 100]内的 80 个年龄随机整数，按区间[0, 10, 18, 35, 50, 70, 100]做 pd.cut()分段统计。

7．分组统计

构造一个至少含有 3 个班级名册的 Excel 成绩表（姓名，学号，班级，性别，课程 1，课程 2）。用 Pandas 读入该表，完成如下计算：

（1）计算每个同学的平均分［平均分 =（课程 1 + 课程 2）/2］。

（2）使用 rank()方法按平均分给出一个全年级的名次排名。

（3）按班级统计每门功课的平均分。

（4）统计男生、女生人数，并给出男生、女生的平均成绩。

8．时间序列

股票每日的涨跌幅在区间[−10%, 10%]内，且只在周一到周五交易。先设置随机数种子为 7，试生成模拟一只股票 2019 年全年交易日的涨跌幅数据。然后计算：

（1）该股票 2019 年的年涨跌幅。

（2）统计 12 个月中涨幅最大的月份和跌幅最大的月份。

实验 8　Matplotlib 绘图库

一、实验目的

（1）学习绘图库 Matplotlib 的使用。

（2）掌握绘图库常见的图形类型，常用的线型、颜色、标记符号。

（3）初步了解 LaTeX 标记。

（4）掌握多子图绘制的方法。

二、实验内容

启动 Spyder，在 Spyder 环境下编写完整程序完成下列练习，部分练习给出了示范代码。

1．利用[2, 3, 5, 10, 8]列表数据绘制折线图、柱形图、饼图。

```
import matplotlib.pyplot as plt                # 导入 plt
plt.rcParams['font.sans-serif'] = ['SimHei']   # 指定中文黑体字体
plt.rcParams['axes.unicode_minus'] = False     # 确保负号显示正常
x = [2, 3, 5, 10, 8]
labels = list('abcde')
plt.plot(x, 'b-')                              # 折线图
plt.figure()                                   # 新建图形
plt.bar(labels, x)                             # 柱形图
plt.figure()                                   # 新建图形
explode = [0, 0, 0.1, 0.2, 0]
plt.pie(x, explode=explode, labels=labels, autopct='%1.1f') # 饼图
plt.show()                                     # 显示图形
```

2．绘出图 12.17 所示的线型和标记颜色符号。

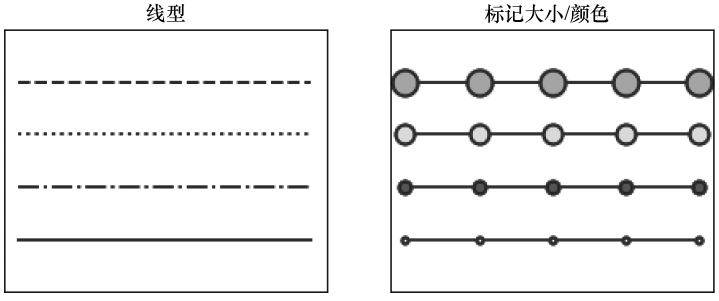


图 12.17　线型和标记

3．绘制一个半径为 2.5 的圆，上半圆用浅绿色填充，下半圆用浅红色填充，如图 12.18 所示。

```
r = 2.5                          # 半径
x = np.linspace(-r, r, 100)      # -r 到 r 之间均匀生成 100 个 x 值
y = np.sqrt(r**2 - x**2)         # 计算圆上点的 y 坐标
plt.plot(x, y, x, -y)            # 画圆。2 组数据，先画上半圆，再画下半圆
plt.fill_between(x, 0, y,  color='g', alpha=0.3)        # 填充上半圆
plt.fill_between(x, 0, -y, color='r', alpha=0.3)        # 填充下半圆
plt.axis('equal')                # 设置横、纵轴单位长度相等
plt.title('圆和填充色')
```

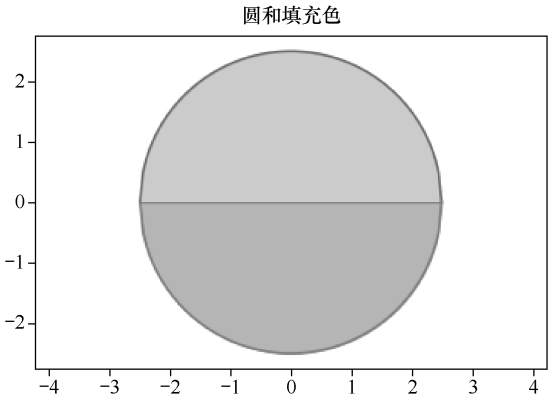


图 12.18　圆和填充色

4．绘制图 12.19 所示的 LaTeX 标记。

```
fig = plt.figure(figsize=(8, 5))      # 图像大小
ax = fig.add_subplot(111)
ax.set_xlim([1, 6])
ax.set_ylim([1, 10])
ax.text(2, 8, r"希腊字母$\alpha\pi\lambda\omega\beta$", fontsize=16)
ax.text(2, 6, r"$ \lim_{x \rightarrow 0} \frac{1}{x} $", fontsize=20)
ax.text(2, 4, r"$x^2+y^2=9$", fontsize=20)
ax.text(2, 2, r"$ \sum_{i=1}^{\infty}\ x_i^2$", fontsize=20)
ax.text(4, 8, r"$ \sin(\frac{\pi}{6})=\frac{1}{2}$", fontsize=20)
ax.text(4, 6, r"$ \sqrt[3]{27} = 3$", fontsize=20)
ax.text(4, 4, r'下标$\alpha_i > \beta_i$',fontsize=14)
ax.text(4, 2, r"$ \int_a^b f(x)dx$", fontsize=20)
ax.text(3, 9, 'LaTeX 示例', fontsize=20)
plt.show()
```

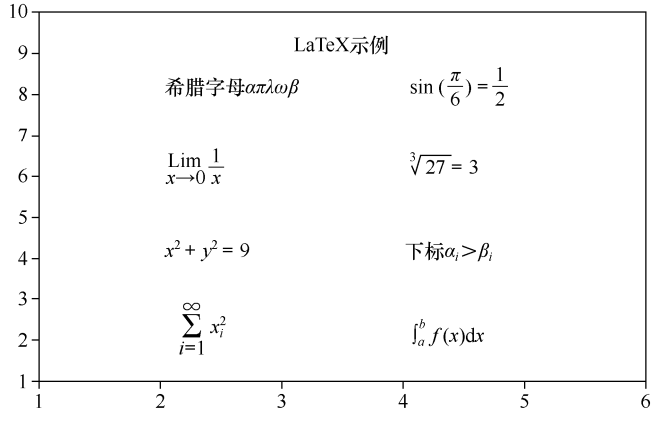


图 12.19　LaTeX 标记示例

5．创建如下数据框，绘制手机销售的对比图形。要求分别用 Pandas 的绘图函数作出饼图和柱形图进行对比。

```
data = {'apple':[1100,1050,1200], 'huawei':[1250,1300,1328], 'oppo':[800,850,750]}
df = DataFrame(data, index=['一月', '二月', '三月'])
```

6．生成 1000 个 $N(1, 10)$正态分布的随机小数，绘制箱线图。

7．绘制如图 12.20 所示的多子图。

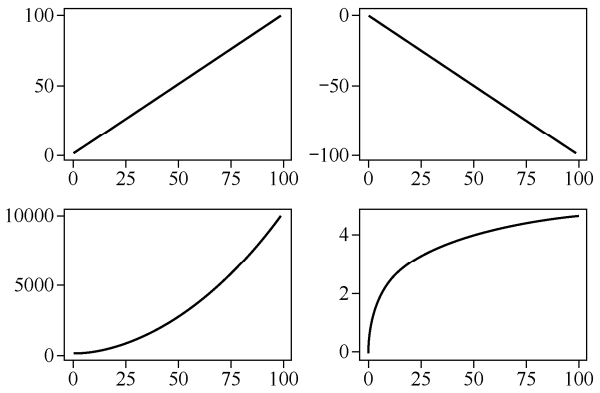


图 12.20　多子图绘制

实验 9　Python 金融数据分析应用

一、实验目的

（1）学习 Python 的日期数据提取与格式变换方法。

（2）学习金融时间序列数据的合并方法。

（3）学习金融时间序列数据可视化图表设置方法。

（4）掌握从世界银行公开数据平台下载我国宏观经济数据的方法。

二、实验内容

1．金融数据序列的日期转换

金融数据的最大特点是时序性，它表明某金融实体在一个特定时期内的数值变动情况。在金融数据序列中比较常见的日期数据格式有字符串对象和 datetime 对象两种。这里以标准模块 datetime 为例说明 Python 日期数据的格式及其操作方法。首先要导入 datetime：

```
In [1]: import datetime
```

（1）获取当天的日期

```
In [2]: datetime.datetime.today()
Out[2]: datetime.datetime(2019, 8, 14, 17, 0, 41, 224187)
```

括号中的前 3 项分别表示年、月、日，其余的数据表示当前的时间。

（2）提取日期中的年、月、日

```
In [3]: datetime.datetime.today().year
Out[3]: 2019
In [4]: datetime.datetime.today().month
Out[4]: 8
In [5]: datetime.datetime.today().day
```

```
Out[5]: 14
```

还可以获得今天是本年度中的第多少周次的星期几。

```
In [6]: datetime.datetime.today().isocalendar()
Out[6]: (2019, 33, 3)
```

输出结果括号中的数据表示今天是 2019 年度第 33 周的周三。这三项数据构成一个元组，所以可用下标变量分别提取每个数据元素。

（3）将 datetime.datetime 对象日期转换为字符串对象日期

```
In [7]: datetime.datetime.today().strftime('%Y-%m-%d')
Out[7]: '2019-08-14'
In [8]: datetime.datetime.today().strftime('%Y%m%d')
Out[8]: '20190814'
```

（4）将字符串对象日期转换为 datetime.datetime 对象日期

```
In [9]: datetime.datetime.strptime('20190814','%Y%m%d')
Out[9]: datetime.datetime(2019, 8, 14, 0, 0)
In [10]: datetime.datetime.strptime('2019-08-14','%Y-%m-%d')
Out[10]: datetime.datetime(2019, 8, 14, 0, 0)
```

2．多个金融数据序列以日期为基准进行整合

合并多个数据序列是金融数据分析工作中经常遇到的事情。当不同数据序列存在不同的数据行或列时，合并过程需要选择合适的轴与合并方向等参数。合并后的数据序列中可能存在缺失值，因此有必要根据分析的需要对缺失值进行处理。下面以合并我国上证指数收盘价格序列与任意单只股票收盘价格序列为例说明两个序列的合并方法。

```
In [11]: start_day = datetime.datetime(1990, 12, 19)  # 上证指数起始日(1990,12,19)
In [12]: end_day = datetime.datetime.today()
In [13]: import  tushare as ts
In [14]: index_data = ts.get_k_data('sh', start_day.strftime(
    '%Y-%m-%d'), end_day.strftime('%Y-%m-%d')).loc[:, ['date', 'close']]  # 上证指数
```

获取单只股票的数据时，这只股票的上市时间未必与上证指数的起始发布时间相同，而且股票自上市以来可能存在停牌（不交易）的情况，所以任何单只股票数据中的日期信息与指数数据中的日期信息会有不同。

```
In [15]: stk_data = ts.get_k_data('601318', start_day.strftime(
    '%Y-%m-%d'), end_day.strftime('%Y-%m-%d')).loc[:, ['date', 'close']]
```

合并两组数据前，重新命名部分列以免发生列名冲突。

```
In [16]: index_data.rename(columns = {'close':'sh_index'},inplace = True)
```

以上证指数的日期列合并单只股票的收盘价格数据。

```
In [17]: import pandas as pd
In [18]:df_merge = pd.merge(
    index_data, stk_data[['date', 'close']], on = 'date', how = 'left')
In [19]: df_merge.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 6839 entries, 0 to 6838
Data columns (total 3 columns):
date       6839 non-null object
close_x    6839 non-null float64
close_y    2971 non-null float64
dtypes: float64(2), object(1)
```

```
memory usage: 213.7+ KB
```

缺失数据在大部分数据分析应用中都很常见。Pandas 使用 NaN（Not a Number）表示浮点和非浮点数组中的缺失数据，它只是一个便于被检测出来的标记而已。

```
In [20]: df_merge.head()
Out[19]:
         date  sh_index  close
0  1990-12-19     113.1    NaN
1  1990-12-20     113.5    NaN
2  1990-12-21     113.5    NaN
3  1990-12-24     114.0    NaN
4  1990-12-25     114.1    NaN
```

对于合并数据表中存在的缺失值 NaN，可以采用回填方法消除。

```
In [21]: df_merge.fillna(method = 'backfill',inplace = True)
In [22]: df_merge.head()
Out[22]:
         date  sh_index  close
0  1990-12-19     113.1  19.89
1  1990-12-20     113.5  19.89
2  1990-12-21     113.5  19.89
3  1990-12-24     114.0  19.89
4  1990-12-25     114.1  19.89
```

这样就可以实现对序列中缺失值的填充（向前或向后要考虑行情数据是按照日期先后排序还是按照日期先后反序排序）。本例向前填充的主要理由是，绝大多数股票的首日上市时间晚于上证指数的首次发布时间，而且股票如果没有交易，那么它的价格是不会变动的，所以向前填充并不影响股票价格走势。

3．金融时间序列可视化图表刻度的设置

在分析金融数据时，常常需要在同一个图表中描述两个或两个以上的金融数据序列。例如，两个纵坐标值差异非常大的数据序列可以在双纵坐标图形中反映两者的时序变换相关性特征，这两个序列甚至可以是有不同单位的数据。

为了实现两个数据序列共享横坐标输出，关键是两个序列要有完全相同的横坐标数据。本实验以上证指数自开市以来各交易日的收盘价格序列与国内市场仍然还在上市交易的单只股票的历史每日收盘价格序列为例，绘制价格走势图形的程序如下：

```
import tushare as ts
import pandas as pd
import datetime
import matplotlib.pyplot as plt
from matplotlib import ticker

start_day = datetime.datetime(1990, 12, 19)  # 上证指数起始日(1990,12,19)
end_day = datetime.datetime.today()
index_data = ts.get_k_data('sh', start_day.strftime('%Y-%m-%d'),
             end_day.strftime('%Y-%m-%d')).loc[:, ['date', 'close']]  # 上证指数
stock_code = '600009'
stk_data = ts.get_k_data(stock_code, start_day.strftime('%Y-%m-%d'),
           end_day.strftime('%Y-%m-%d')).loc[:, ['date', 'close']]
```

```
index_data.rename(columns = {'close': 'sh_index'}, inplace = True)
df_merge = pd.merge(
    index_data, stk_data[['date', 'close']], on = 'date', how = 'left')
df_merge.fillna(method = 'backfill', inplace = True)

def format_date(x, pos):  # 设置坐标格式
    if x < 0 or x > len(date_tickers)-1:
        return ''
    return date_tickers[int(x)]
date_tickers = df_merge['date'].values
xmajorLocator = ticker.MultipleLocator(220)
plt.rcParams['font.sans-serif'] = ['SimHei']  # 用来正常显示中文标签
fig, ax = plt.subplots(figsize=(9, 6))
plt.xticks(df_merge.index, df_merge.date)
plt.xticks(rotation=50)
ax.plot(df_merge.index, df_merge['sh_index'], '-r', lw = 1.5, label = '上证指数')
plt.legend(loc = 'upper left')
ax.xaxis.set_major_formatter(ticker.FuncFormatter(format_date))
ax.xaxis.set_major_locator(xmajorLocator)
ax2 = ax.twinx()  # 这是双坐标的关键一步
ax2.plot(df_merge.index, df_merge['close'], '-y', lw = 1.5, label = stock_code)
plt.legend(loc = 'upper right')
plt.title('市场指数与个股价格走势')
plt.grid(True)
plt.show()
```

由于两组序列数据量非常大（包含 6800 多个交易日的数据），如果不对横坐标（交易日）进行格式设置，那么输出图形的横坐标标签将是一片黑影。为此，本程序定义了一个用于横坐标刻度格式的函数 `format_date()`，且利用 Matplotlib 包的 `ticker.MultipleLocator()`设定横坐标显示的刻度数量。程序运行的输出图形如图 12.21 所示。

图 12.21　数据序列双轴图

4．世界银行公开数据下载步骤

（1）在浏览器地址栏输入网址 https://data.worldbank.org.cn/，打开如图 12.22 所示的世界银行公开数据平台首页。单击网页左上方的“从这里开始”超链接，可以打开关于获取公开数据的简要操作指引页面，大致了解如何从平台查找数据的方法。

图 12.22　世界银行公开数据平台首页

（2）单击如图 12.23 所示网页右上角的菜单图标（见图中的红圈标记处）打开浏览国家和指标菜单列表（见图 12.24）。单击“国家”超链接打开如图 12.25 所示的页面。

图 12.23　菜单列表图标

（3）单击网页左上角的“国家”，显示可查询的全部国家名称的第一个字名。

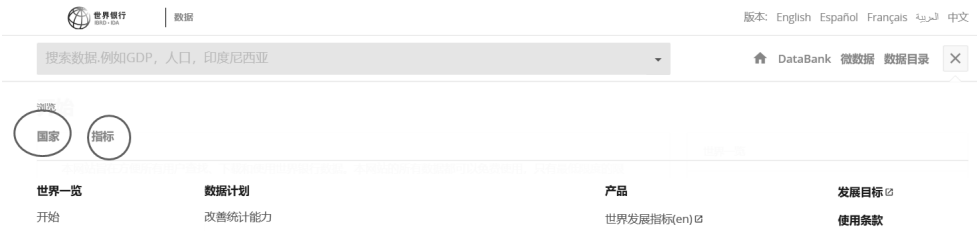


图 12.24　菜单列表

（4）单击如图 12.25 所示列表上的“中”超链接，显示所有以“中”开头的国家或地区经济体的名称列表（见图 12.26）。

（5）单击如图 12.26 中的“中国”超链接，打开查询中国经济发展数据的页面（见图 12.27）。

图 12.25　国家名称首个汉字名表

图 12.26　以“中”字开头的经济体名表

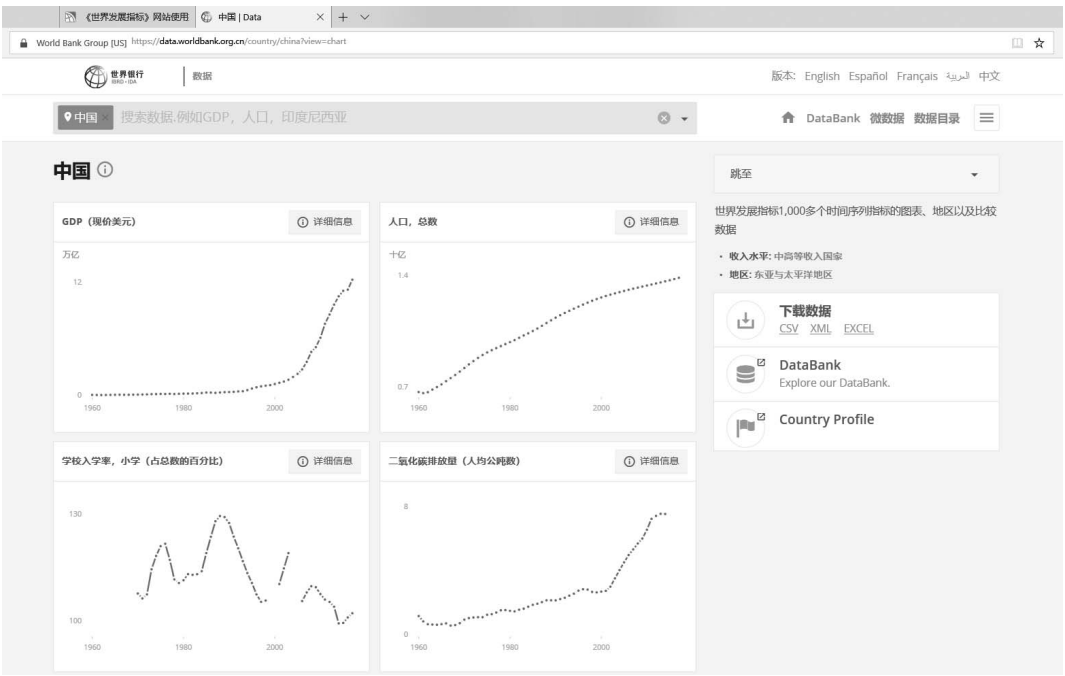


图 12.27　经济发展指标例图

（6）单击如图 12.27 中的任一图表［如 GDP（现价美元）图表］左上角的“详细信息”，接着单击如图 12.28 所示对话框中的“所有元数据”，打开数据查询页面（见图 12.29）。

图 12.28　详细信息对话框

（7）在如图 12.29 所示的页面中依次单击 Country、Series 和 Time，分别选择国家列表中的“中国”（见图 12.29）、计划下载的宏观经济指标数据序列（见图 12.30）及经济指标数据序列的时间范围（见图 12.31），且每次选择后都要在如图 12.29 所示的弹出对话框中单击“应用更改”按钮以刷新选择结果。

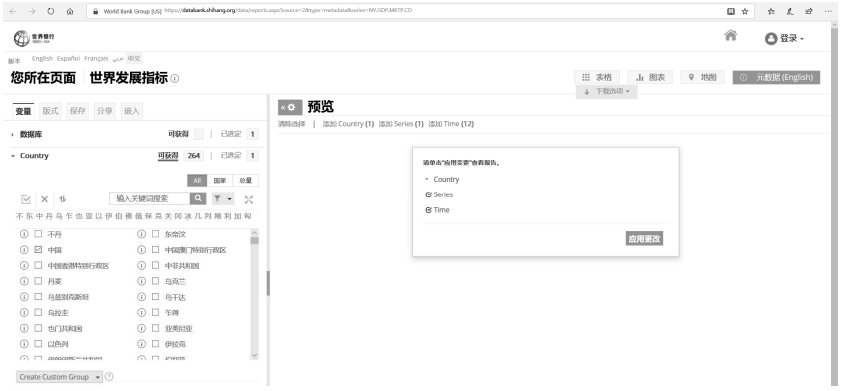


图 12.29　选择国家或地区

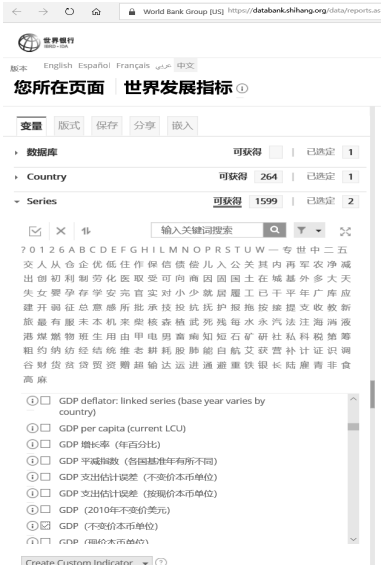


图 12.30　选择数据序列

图 12.31　选择时间

（8）单击如图 12.32 所示页面右上角的“下载选项”，确定下载数据保存的文件类型。建议选择保存为 CSV 文件，因为 CSV 文件数据比较方便 Python 程序读取。

图 12.32　下载选项

参 考 文 献

[1] 张若愚. Python 科学计算（第 2 版）. 北京：清华大学出版社，2016.

[2] 董付国. Python 程序设计基础（第 2 版）. 北京：清华大学出版社，2018.

[3] 江红，余青松. Python 程序设计与算法基础教程（第 2 版）. 北京：清华大学出版社，2019.

[4] 李刚. 疯狂 Python 讲义. 北京：电子工业出版社，2019.

[5] 阿布. 量化交易之路——用 Python 做股票量化分析. 北京：机械工业出版社，2018.

[6] Joel Grus. 数据科学入门. 高蓉，韩波译. 北京：人民邮电出版社，2016.

[7] Wes McKinney. *Python for Data Analysis*, 2nd Edition. Sebastopol: O'REILLY, 2018.

[8] Ivan Idris. NumPy 学习指南（第 2 版）. 张驭宇译. 北京：人民邮电出版社，2014.

[9] David Beazley. Python Cookbook（第 3 版）中文版. 陈舸译. 北京：人民邮电出版社，2015.

[10] Wesley-Chun. Python 核心编程（第 3 版）中文版. 孙波翔等译. 北京：人民邮电出版社，2016.

[11] James Payne. Python 编程入门经典. 张春晖译. 北京：清华大学出版社，2010.

[12] Yves Hilpisch. Python 金融大数据分析. 姚军译. 北京：人民邮电出版社，2015.

[13] John Zelle. Python 程序设计（第 3 版）. 王海鹏译. 北京：人民邮电出版社，2018.